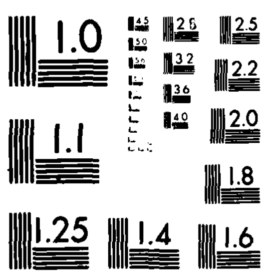
AD-A108 527 SOFTECH INC WALTHAM MA F/G 5/9
THE JOVIAL (J73) WORKBOOK. VOLUME I. INTEGER AND FLOATING POINT--ETC(U)
NOV 81 F30602-79-C-0040
UNCLASSIFIED RADC-TR-81-333-VOL-1 NL

1 of 5

SIFIED

1 OF 5

AD
A108527



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

PHOTOGRAPH THIS SHEET

ADA108527

DTIC ACCESSION NUMBER

LEVEL

INVENTORY

Softech, Inc.
Waltham, MA
The JOVIAL (J73) Workbook, Vols I-V
Nov. 81

DOCUMENT IDENTIFICATION

Contract F30602-79-C-0040 Rept.No RADC-TR-81-333 Vols. I-VI (of 15)

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DISTRIBUTION STATEMENT

ACCESSION FOR		
NTIS	GRA&I	☒
DTIC	TAB	☐
UNANNOUNCED		☐
JUSTIFICATION		
BY		
DISTRIBUTION /		
AVAILABILITY CODES		
DIST	AVAIL AND/OR SPECIAL	
A		

DISTRIBUTION STAMP

DTIC
ELECTE
DEC 14 1981
S D
D

DATE ACCESSIONED

DATE RECEIVED IN DTIC

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-DDA-2

DTIC FORM OCT 79 70A

DOCUMENT PROCESSING SHEET

RADC-TR-81-333, Vols I-VI (of 15)
Interim Report
November 1981

THE JOVIAL (J73) WORKBOOK

SofTech, Inc.

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-81-333, Vols I-VI (of 15) have been reviewed and are approved for publication.

APPROVED: Douglas A. White

DOUGLAS A. WHITE
Project Engineer

APPROVED: J J Marciniak

JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: RADC-TR-81-333, Vols I-VI (of 15)
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:
4. TITLE (and Subtitle): THE JOVIAL (J73) WORKBOOK
5. TYPE OF REPORT & PERIOD COVERED: Interim Report, Dec 79 - Oct 81
6. PERFORMING ORG. REPORT NUMBER: N/A
7. AUTHOR(s): N/A
8. CONTRACT OR GRANT NUMBER(s): F30602-79-C-0040
9. PERFORMING ORGANIZATION NAME AND ADDRESS: SofTech, Inc., 460 Totten Pond Rd, Waltham MA 02154
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 33126F, 20220403
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIS), Griffiss AFB NY 13441
12. REPORT DATE: November 1981
13. NUMBER OF PAGES: 303
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same
15. SECURITY CLASS. (of this report): UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report): Approved for public release; distribution unlimited.
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report): Same
18. SUPPLEMENTARY NOTES: RADC Project Engineer: Douglas A. White (ISIS)
19. KEY WORDS (Continue on reverse side if necessary and identify by block number):
    JOVIAL (J73)
    MIL-STD-1589A
    Video Course
    Higher Order Language
20. ABSTRACT (Continue on reverse side if necessary and identify by block number):

The JOVIAL (J73) Workbook is only one portion of a self-instructional JOVIAL (J73) training course. In addition to the programmed-learning primer/workbooks, are video taped lectures. The workbooks are formatted to consist of fifteen (15) segments bound in three (3) volumes covering each particular language capability. A video tape lecture was prepared for each workbook segment. This course is taught in two parts. Part I contains twelve (12) segments in Volumes I and II of the workbook; Part II

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

THE JOVIAL (J73) WORKBOOK

VOLUME 1

INTEGER AND FLOATING POINT ITEM-DECLARATIONS

1081-1

April 1981

Submitted to

Department of the Air Force
Rome Air Development Center
I S I S
Griffiss Air Force Base, NY 13441

Prepared by

SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02154

PREFACE

This workbook is intended for use with Tape 1 of the JOVIAL (J73) Video Course. Its purpose is to elaborate upon and reinforce concepts and language rules introduced in the videotape.

Following a brief introduction to High Order Languages and the history of JOVIAL (J73), an overview of a complete JOVIAL (J73) program is presented. Integer and floating point item-declarations are then discussed in detail, followed by their use in assignment statements and formulae. The final pages are a summary of the information presented in the first lesson.

TABLE OF CONTENTS

SECTION 1

INTRODUCTION

SOFTECH

WHY HIGH ORDER LANGUAGES?

High Order Languages (HOLs) permit programmers to write software in an English-like expression-oriented notation, rather than in the machine-oriented assembly languages actually "understood" by their computers. For example, the operation coded by the HOL programmer as:

SET C = A + B

might require the assembly language programmer to write:

LD A

AD B

ST C

There are many advantages to the use of HOLs. Programs are easier to read, reducing debugging and maintenance costs. They are also easier to write, because programmers can learn HOLs more easily, express their algorithms more naturally, and produce code more quickly. Thus life cycle costs are lower.

HOls have been used almost exclusively in business programming for many years, but assembly language is still heavily used for many state-of-the-art scientific and military applications. There are several reasons for this:

- HOLs are not always available for the selected computer.
- HOLs may not provide access to special features of the computer.
- Use of HOLs may make it difficult to obtain the desired software efficiency - programs may occupy more memory or run more slowly than those written in assembly language.

Modern developments in the design of HOLs and their translators are overcoming many of these difficulties, and making the benefits of HOL usage available to a wider variety of applications. Languages are being designed to provide the needed capabilities, language translators are employing sophisticated optimization techniques to produce more efficient code, and language standardization is increasing the potential for software reuse.

The following chart summarizes the three levels of programming languages.

MACHINE LANGUAGE	ACTUAL BINARY INSTRUCTIONS EXECUTED ONE AT A TIME BY THE COMPUTER: 01101100 10001101 . . .
ASSEMBLY LANGUAGE	MNEMONIC INSTRUCTIONS TRANSLATED ONE-FOR-ONE TO MACHINE LANGUAGE EQUIVALENTS (BY AN ASSEMBLER): LD A AD B ST C . . .
HIGH ORDER LANGUAGE (HOL)	MORE ENGLISH-LIKE INSTRUCTIONS TRANSLATED ONE-TO-MANY TO MACHINE LANGUAGE EQUIVALENTS (BY A COMPILER): SET C = A + B; . . .

WHY A COMMON LANGUAGE?

There are presently many hundreds of HOLs in use for the development of applications software. These different HOLs have been developed to meet the specific needs of different applications, or to incorporate the ideas and philosophies of different language designers. Many of these HOLs are available on only a few kinds of computers.

This activity has led to many advances in language design, but it has had significant costs in translator development and programmer retraining. In recent years there has been an increasing interest in language standardization, particularly within the Department of Defense. This standardization will result in significant cost savings in translator development and in programming training. An even more important cost savings will come from software reusability.

When all software is written in the same language, it can be more easily reused from one project to the next. Many systems perform a number of functions in common, affording many opportunities for software reuse. For example, all on-board aircraft systems will contain software to communicate with navigation systems. If such software is written in a common language, it can be reused from one aircraft system to another. Because the common HOL will be available on many different computers, this is true even when the systems use different computers. (This is called software portability.) The DoD expects to realise substantial cost savings through reuse of software written in a standard common language.

THE DEVELOPMENT OF JOVIAL (J73)

The J73 language is the latest in an evolving JOVIAL family. JOVIAL J3, the first widely used JOVIAL dialect, was used in many military systems during the 1960's. Two dialects of J3 were designed independently in the late 1960's and early 1970's. The Boeing B-1 program developed the J3B language, and the Air Force developed a standard JOVIAL called J73, of which only a first level subset, J73/I, was implemented.

When the DoD decided to establish interim standards for an HOL, the Air Force programming community chose to combine the best features of J3B and J73/I, to form a new language, also called J73. SofTech was awarded a contract to perform this language design and to develop the first set of J73 compilers. Several ongoing Air Force and Army programs are now using J73. Figure 1-1 illustrates the J73 evolution.

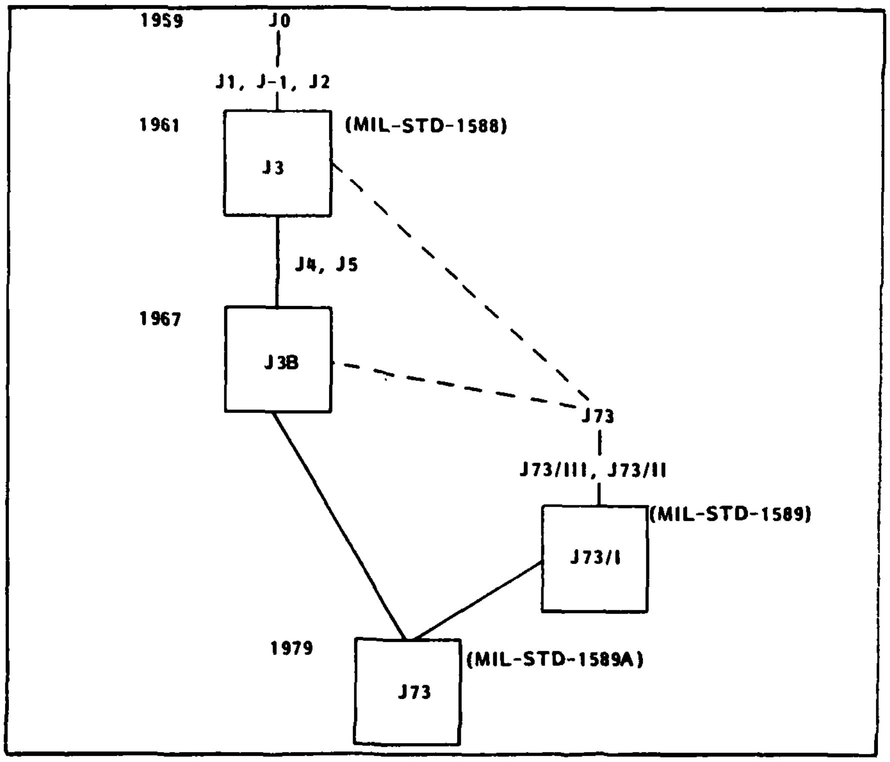


Figure 1-1. The JOVIAL Language Family

FEATURES OF JOVIAL (J73)

- Modular constructs for "block-structured" programs - this avoids multiple GOTOs.
- Structured control-flow statements such as loops, IF, and CASE; restrictions on GOTO statement.
- Strong type checking to ensure that data will be used as intended.
- Machine-specific functions and specified tables for low-level operations and control of storage.
- Machine parameters for portability.

JOVIAL (J73) is defined by MIL-STD-1589A (USAF) 15 March 1979. SofTech has also published the Computer Programming Manual for the JOVIAL (J73) Language. These documents should be used for further reference.

SECTION 2

PROGRAM OVERVIEW

SOFTECH

PROGRAM OVERVIEW

While every JOVIAL (J73) program has characteristics specific to its algorithmic purpose, there are certain elements that have general application. A brief look at these elements in the context of an actual program will provide a useful starting point for a more detailed discussion of the language.

Figure 2-1 is a sample of a complete JOVIAL (J73) program written in one module (compilation unit). This program contains in one place all the information necessary to perform a given algorithm. Notice that this module is surrounded by a START and TERM. All JOVIAL (J73) modules are delineated in this manner. Since this is a main program module, the program is also given a name. A BEGIN/END pair surround the program body.

A program must also have a **name** (see Figure 2-2). A name is a sequence of letters, digits, dollar signs and primes. It must begin with a letter or dollar sign, and it must be at least two characters long. A JOVIAL (J73) name can be any length, though the compiler only looks at the first 31 characters (depending upon the implementation). Names are discussed in more detail below.

Declarations appear at the beginning of each program and associate names with programmer-supplied meanings.

> *NOTE: Except for statement-names, names of subroutines, type names in point-item-descriptions, and formal parameter names, a name may not be used prior to the point at which a declaration for that name appears.*

```
START "PROGRAM MODULE"
PROGRAM PERFECT'NUMBER ;

"THIS PROGRAM FINDS ALL PERFECT NUMBERS IN THE RANGE 2..1000"
"A PERFECT NUMBER IS A NUMBER THAT IS EQUAL TO THE SUM OF
      ALL ITS FACTORS EXCEPT ITSELF"

      BEGIN "PERFECT NUMBER"

      "DECLARATIONS"

      TABLE FACTAB (1 : 500) ;
            ITEM FACTOR B ;
      TABLE PERFECTAB (1 : 1000) ;
            ITEM PERFECT B ;
      ITEM SUM U ;

      "EXECUTION - FOR EACH INTEGER IN THE RANGE 2..1000 CALL A SUBROUTINE
            TO FIND THE FACTORS, SUM THE FACTORS, TEST, IF PERFECT FLAG TRUE"

      FOR I : 2 BY 1 WHILE I <= 1000 ;
            BEGIN "FOR LOOP"
            FINDFACTOR (I : FACTAB) ;     "SUBROUTINE CALL"
            SUM = 0 ;
            FOR J : 1 BY 1 WHILE J < I ;  "SUM THE FACTORS"
                  IF FACTOR (J) ;
                        SUM = SUM + J;
            IF SUM = I ;                  "TEST IF PERFECT"
                  PERFECT (I) = TRUE ;    "FLAG IF PERFECT"
            END "FOR LOOP"

      PROC FINDFACTOR (NUMBER : FACTORTAB) :

      "THIS PROCEDURE TAKES ANY NUMBER AS INPUT. BUT WILL ONLY EXECUTE
            CORRECTLY FOR NUMBERS IN THE RANGE 1..1000 - THE PROCEDURE
            OUTPUTS A TABLE OF FACTORS IN THE RANGE 1..NUMBER/2"

            BEGIN "PROC FINDFACTOR"

            "DECLARATION OF FORMAL PARAMETERS"

            ITEM NUMBER U ;
            TABLE FACTORTAB (1 : 500);
                  ITEM FACTORBIT B;

            "EXECUTION - INITIALIZE ALL FACTORS FALSE, FOR EACH INTEGER IN THE
                  RANGE 1..SQUARE ROOT OF INTEGER, TEST IF IT IS A FACTOR, IF IT
                  IS, FLAG THE BIT TRUE AND FLAG ITS CO-FACTOR TRUE;

            FOR I : 1 BY 1 WHILE I <= 500 ;  "INITIALIZE"
                  FACTORBIT (I) = FALSE ;

            FOR I : 1 BY 1 WHILE I <= (* U *) (NUMBER ** .5);
                  IF NUMBER MOD I = 0;     "TEST IF REMAINDER = 0, THEN A FACTOR"
                        BEGIN "SET FACTOR BITS"
                        FACTORBIT (I) = TRUE ;
                        FACTORBIT (NUMER / I) = TRUE;
                        END "SET FACTOR BITS"
            END "PROC FINDFACTOR"
      END "PERFECT'NUMBER"
TERM
```

Figure 2-1. START/TERM and BEGIN/END Program Delineators

```
START "PROGRAM MODULE"
PROGRAM PERFECT'NUMBER ;

"THIS PROGRAM FINDS ALL PERFECT NUMBERS IN THE RANGE 2..1000"
"A PERFECT NUMBER IS A NUMBER THAT IS EQUAL TO THE SUM OF
      ALL ITS FACTORS EXCEPT ITSELF"

      BEGIN "PERFECT NUMBER"

      "DECLARATIONS"

      TABLE FACTAB (1 : 500) ;
            ITEM FACTOR B ;
      TABLE PERFECTAB (1 : 1000) ;
            ITEM PERFECT B ;
      ITEM SUM U ;

      "EXECUTION - FOR EACH INTEGER IN THE RANGE 2..1000 CALL A SUBROUTINE
            TO FIND THE FACTORS, SUM THE FACTORS, TEST, IF PERFECT FLAG TRUE"

      FOR I : 2 BY 1 WHILE I <= 1000 ;
            BEGIN "FOR LOOP"
            FINDFACTOR (I : FACTAB) ;      "SUBROUTINE CALL"
            SUM = 0 ;
            FOR J : 1 BY 1 WHILE J < I ;   "SUM THE FACTORS"
                  IF FACTOR (J) ;
                        SUM = SUM + J;
            IF SUM = I ;                   "TEST IF PERFECT"
                  PERFECT (I) = TRUE ;     "FLAG IF PERFECT"
            END "FOR LOOP"

      PROC FINDFACTOR (NUMBER : FACTORTAB) :

      "THIS PROCEDURE TAKES ANY NUMBER AS INPUT, BUT WILL ONLY EXECUTE
            CORRECTLY FOR NUMBERS IN THE RANGE 1..1000 - THE PROCEDURE
            OUTPUTS A TABLE OF FACTORS IN THE RANGE 1..NUMBER/2"

            BEGIN "PROC FINDFACTOR"

            "DECLARATION OF FORMAL PARAMETERS"

            ITEM NUMBER U ;
            TABLE FACTORTAB (1 : 500);
                  ITEM FACTORBIT B;

            "EXECUTION - INITIALIZE ALL FACTORS FALSE, FOR EACH INTEGER IN THE
                  RANGE 1..SQUARE ROOT OF INTEGER, TEST IF IT IS A FACTOR, IF IT
                  IS, FLAG THE BIT TRUE AND FLAG ITS CO-FACTOR TRUE;

            FOR I : 1 BY 1 WHILE I <= 500 ;  "INITIALIZE"
                  FACTORBIT (I) = FALSE ;

            FOR I : 1 BY 1 WHILE I <= (* U *) (NUMBER ** .5);
                  IF NUMBER MOD I = 0;       "TEST IF REMAINDER = 0, THEN A FACTOR"
                        BEGIN "SET FACTOR BITS"
                        FACTORBIT (I) = TRUE ;
                        FACTORBIT (NUMER / I) = TRUE;
                        END "SET FACTOR BITS"
            END "PROC FINDFACTOR"
      END "PERFECT'NUMBER"
TERM
```

Figure 2-2. JOVIAL (J73) Program Name

Data-declarations declare data-names and their attributes (Figure 2-3). There are three kinds of data structures in J73:

- Item - A simple data object of the language. An item is a variable of a pre-defined or programmer-defined type having no constituents.
- Table - An aggregate data object consisting of a collection of one or more items, or an array of such collections.
- Block - A group of items and tables and other blocks allocated a contiguous area of storage.

Items and tables may be variable or constant; blocks may contain constant items and tables. The declaration of a variable data object provides information that is used when it is allocated storage in memory. Variable data objects may be preset (given an initial value), referenced (read), changed (written) and possibly deallocated. A constant data object is preset to its constant value; it may only be referenced (read). In some cases, the value of a constant data object may be allocated storage in memory, but in many cases the value is embedded into the code of the compiled program.

The compiler allocates data objects on a word by word basis (unless otherwise specified, using the advanced features of the language). This means that each component of each data object is allocated a new word.

In order for a J73 program to actually "do" anything, it must contain **executable statements** (see Figure 2-4). Statements are the means by which computational algorithms are specified. They control the execution of the complete program. There are ten varieties of executable statements in J73:

- assignment statements
- loop statements
- IF statements
- CASE statements
- procedure call statements

```
START "PROGRAM MODULE"
PROGRAM PERFECT'NUMBER ;

"THIS PROGRAM FINDS ALL PERFECT NUMBERS IN THE RANGE 2..1000"
"A PERFECT NUMBER IS A NUMBER THAT IS EQUAL TO THE SUM OF
      ALL ITS FACTORS EXCEPT ITSELF"

      BEGIN "PERFECT NUMBER"

      "DECLARATIONS"

      TABLE FACTAB (1 : 500) ;
            ITEM FACTOR B ;
      TABLE PERFECTAB (1 : 1000) ;
            ITEM PERFECT B ;
      ITEM SUM U ;

      "EXECUTION - FOR EACH INTEGER IN THE RANGE 2..1000 CALL A SUBROUTINE
            TO FIND THE FACTORS, SUM THE FACTORS, TEST, IF PERFECT FLAG TRUE"

      FOR I : 2 BY 1 WHILE I <= 1000 ;
            BEGIN "FOR LOOP"
            FINDFACTOR (I : FACTAB) ;       "SUBROUTINE CALL"
            SUM = 0 ;
            FOR J : 1 BY 1 WHILE J < I ;    "SUM THE FACTORS"
                  IF FACTOR (J) ;
                        SUM = SUM + J;
            IF SUM = I ;                    "TEST IF PERFECT"
                  PERFECT (I) = TRUE ;      "FLAG IF PERFECT"
            END "FOR LOOP"

      PROC FINDFACTOR (NUMBER : FACTORTAB) :

      "THIS PROCEDURE TAKES ANY NUMBER AS INPUT, BUT WILL ONLY EXECUTE
            CORRECTLY FOR NUMBERS IN THE RANGE 1..1000 - THE PROCEDURE
            OUTPUTS A TABLE OF FACTORS IN THE RANGE 1..NUMBER/2"

            BEGIN "PROC FINDFACTOR"

            "DECLARATION OF FORMAL PARAMETERS"

            ITEM NUMBER U ;
            TABLE FACTORTAB (1 : 500);
                  ITEM FACTORBIT B;

            "EXECUTION - INITIALIZE ALL FACTORS FALSE, FOR EACH INTEGER IN THE
                  RANGE 1..SQUARE ROOT OF INTEGER, TEST IF IT IS A FACTOR, IF IT
                  IS, FLAG THE BIT TRUE AND FLAG ITS CO-FACTOR TRUE;

            FOR I : 1 BY 1 WHILE I <= 500 ;  "INITIALIZE"
                  FACTORBIT (I) = FALSE ;

            FOR I : 1 BY 1 WHILE I <= (* U *) (NUMBER ** .5);
                  IF NUMBER MOD I = 0;      "TEST IF REMAINDER = 0, THEN A FACTOR"
                        BEGIN "SET FACTOR BITS"
                        FACTORBIT (I) = TRUE ;
                        FACTORBIT (NUMER / I) = TRUE;
                        END "SET FACTOR BITS"
            END "PROC FINDFACTOR"
      END "PERFECT'NUMBER"
TERM
```

Figure 2-3. J73 Main Program & Subroutine Data Declarations

```
START "PROGRAM MODULE"
PROGRAM PERFECT'NUMBER ;

"THIS PROGRAM FINDS ALL PERFECT NUMBERS IN THE RANGE 2..1000"
"A PERFECT NUMBER IS A NUMBER THAT IS EQUAL TO THE SUM OF
      ALL ITS FACTORS EXCEPT ITSELF"

      BEGIN "PERFECT NUMBER"

      "DECLARATIONS"

      TABLE FACTAB (1 : 500) ;
            ITEM FACTOR B ;
      TABLE PERFECTAB (1 : 1000) ;
            ITEM PERFECT B ;
      ITEM SUM U ;

      "EXECUTION - FOR EACH INTEGER IN THE RANGE 2..1000 CALL A SUBROUTINE
            TO FIND THE FACTORS, SUM THE FACTORS, TEST, IF PERFECT FLAG TRUE"

      FOR I : 2 BY 1 WHILE I <= 1000 ;
            BEGIN "FOR LOOP"
            FINDFACTOR (I : FACTAB) ;      "SUBROUTINE CALL"
            SUM = 0 ;
            FOR J : 1 BY 1 WHILE J < I ;   "SUM THE FACTORS"
                  IF FACTOR (J) ;
                        SUM = SUM + J;
            IF SUM = I ;                   "TEST IF PERFECT"
                  PERFECT (I) = TRUE ;     "FLAG IF PERFECT"
            END "FOR LOOP"

      PROC FINDFACTOR (NUMBER : FACTORTAB) :

      "THIS PROCEDURE TAKES ANY NUMBER AS INPUT, BUT WILL ONLY EXECUTE
            CORRECTLY FOR NUMBERS IN THE RANGE 1..1000 - THE PROCEDURE
            OUTPUTS A TABLE OF FACTORS IN THE RANGE 1..NUMBER/2"

            BEGIN "PROC FINDFACTOR"

            "DECLARATION OF FORMAL PARAMETERS"

            ITEM NUMBER U ;
            TABLE FACTORTAB (1 : 500);
                  ITEM FACTORBIT B;

            "EXECUTION - INITIALIZE ALL FACTORS FALSE, FOR EACH INTEGER IN THE
                  RANGE 1..SQUARE ROOT OF INTEGER, TEST IF IT IS A FACTOR, IF IT
                  IS, FLAG THE BIT TRUE AND FLAG ITS CO-FACTOR TRUE;

            FOR I : 1 BY 1 WHILE I <= 500 ;   "INITIALIZE"
                  FACTORBIT (I) = FALSE ;

            FOR I : 1 BY 1 WHILE I <= (* U *) (NUMBER ** .5);
                  IF NUMBER MOD I = 0;      "TEST IF REMAINDER = 0, THEN A FACTOR"
                        BEGIN "SET FACTOR BITS"
                        FACTORBIT (I) = TRUE ;
                        FACTORBIT (NUMER / I) = TRUE;
                        END "SET FACTOR BITS"
            END "PROC FINDFACTOR"
      END "PERFECT'NUMBER"
TERM
```

Figure 2-4. Executable Statements

- RETURN statements
- GOTO statements
- EXIT statements
- STOP statements
- ABORT statements

These will be introduced gradually during the course.

Subroutines to be used in the program may be defined following the executable statements of the main program. Subroutines are like small programs that are executed when they are called. If the same group of executable statements is needed at more than one point in the program, or if these executable statements perform a single, logical function, a subroutine that contains those statements may be written and then called from anywhere in the program. In the body of the subroutine, as in the body of the program itself, declarations appear first, followed by executable statements (see Figures 2-3 and 2-4).

SYNTAX

The syntax conventions used in the JOVIAL (J73) Video Course and workbook are as follows:

<table>
<tr><th>Syntax</th><th>Meaning</th><th>Sample Expansions</th></tr>
<tr><td>[some-feature]</td><td>Brackets indicate an optional feature.</td><td>some-feature
OR
nothing</td></tr>
<tr><td>{one | other}</td><td>Braces with a vertical bar indicate disjunction- a choice between alternatives.</td><td>one
OR
other</td></tr>
<tr><td>{this-one
that-one}</td><td>Braces with each feature on separate lines indicate disjunction - a choice between alternatives.</td><td>this-one
OR
that-one</td></tr>
<tr><td>letter ...</td><td>The sequence '...' indicates one or more repetitions of a feature.</td><td>letter
letter letter
letter letter letter
...</td></tr>
<tr><td>(letter),...</td><td>The sequence "..." following a comma (or a colon) indicates one or more repetitions of the feature separated by commas (or colons).</td><td>(letter)
(letter) (letter)
(letter) (letter) (letter)
...</td></tr>
<tr><td>[{this-one
that-one}]
+ another</td><td>Syntax symbols may be combined.</td><td>this-one + (another)
that-one + (another)
+ (another)</td></tr>
</table>

SECTION 3

DATA-DECLARATIONS: INTEGER AND FLOATING POINT

DATA-DECLARATIONS: INTEGER AND FLOATING POINT

As mentioned in the previous section, data-declarations declare data-names and their attributes for the three kinds of J73 data structures-items, tables and blocks. This section begins to address the first of these structures -- **items**.

There are seven kinds of items in JOVIAL (J73):

- integer (signed and unsigned) - signed integer items are used for non-negative and negative whole number values; unsigned integer items are used for non-negative whole number values only.
- floating point - used for fractional values
- fixed point - used for fractional values
- bit - used for a string of bits
- character - used for a string of characters (bytes)
- status - allows all possible values of an item to be enumerated
- pointer - used for an item which contains as its value a machine address

The same general rules of syntax apply to each type (see Figure 3-1), though only integer and floating point items will be discussed in this first workbook.

```
                                 ┌ type-description ┐
[CONSTANT] ITEM name  ┤                  ├  [item-preset];
                                 └ item-type-name   ┘
```

Figure 3-1. General Syntax for Item Data Declarations

ITEM ITEM is a JOVIAL (J73) reserved word. It has a specific meaning to the compiler, which restricts its use to declaration. It cannot be used as a name.

NAME A name is a sequence of letters, digits, dollar signs and primes (single quote characters). It must begin with a letter or a dollar sign, and be at least two characters long. A name may be any length, though only the first thirty-one characters are examined for uniqueness within the program. (Fewer than thirty-one characters may be examined for external names. External names are discussed in Tape 8.*

; All JOVIAL (J73) statements must end with a semi-colon.

---

*The following are JOVIAL (J73) reserved words, and cannot be used as names:

```
    ABORT | ABS | AND | BEGIN | BIT
  | BITSIZE | BLOCK | BY | BYTE
  | BYTESIZE | CASE | COMPOOL
  | CONSTANT | DEF | DEFAULT
  | DEFINE | ELSE | END | EQV
  | FALLTHRU | FALSE | FIRST | FOR
  | GOTO | IF | INLINE | INSTANCE
  | ITEM | LABEL | LAST | LBOUND
  | LIKE | LOC | MOD | NEXT | NOT
  | NULL | NWDSEN | OR | OVERLAY
  | PARALLEL | POS | PROC | PROGRAM
  | REC | REF | RENT | REP | RETURN
  | SGN | SHIFTL | SHIFTR | START
  | STATIC | STATUS | STOP | TABLE
  | TERM | THEN | TRUE | UBOUND
  | WHILE | WORDSIZE | XOR
```

INTEGER ITEM-DECLARATIONS

An integer is a positive or negative whole number value. Data objects which are to be used as integers must be so declared at the beginning of a program or subroutine. In its simplest form, an integer item-declaration might look like this --

ITEM	name	type-description	semi-colon
↓	↓	↓	↓
ITEM	SUM	S 15	;

ITEM JOVIAL (J73) reserved word.

name A legal JOVIAL (J73) name (unique and supplied by the programmer).

type-description The type-description for an integer consists of a single letter type indicator and an optional integer-size. The type indicators are:

U - an unsigned integer type used to represent non-negative number values

S - A signed integer type, used to represent both negative and non-negative whole number values.

Integer-size specifies the minimum number of bits of storage to be allocated to hold the value of the integer item. If integer-size is omitted, it defaults to BITSINWORD-1 (a single word integer). (BITSINWORD is an implementation parameter equal to the number of bits in a JOVIAL (J73) word.)

; A semi-colon terminates the declaration.

Thus, ITEM SUM S 15; declares an item SUM to be a signed integer item allocated at least 15 bits of storage for its value and one sign bit. Consider the following examples:

ITEM name	type-description	semi-colon	
↓ ↓	↓	↓	
ITEM SPEED	U 10	;	SPEED is an unsigned integer variable allocated 10 bits of storage for its non-negative integer value
ITEM DISTANCE	S 6	;	DISTANCE is a signed integer variable allocated 6 bits of storage for its non-negative or negative integer value and one sign bit
ITEM AREA	U	;	AREA is an unsigned integer variable allocated 1 word of storage (including sign bit). If BITSINWORD is 16 area is allocated 15 bits for its value
ITEM METERS	S	;	METERS is a signed integer variable allocated 1 word of storage (including sign bit). If BITSINWORD is 32, meters is allocated 31 bits for its value.

NOTES: In many cases, the programmer can choose to ignore the number of bits allocated for an integer item and use the compiler default. A single-word integer is sufficiently large for many computations. There are two cases where integer-size must be specified:

- *When memory space it sight and more than one item is to share a word of memory, integer-size needs to be specified.*
- *When the values to be represented in an integer item are larger than can be represented in one word, a double-word integer-size needs to be specified.*

If BITSINWORD is a 16, a U 16 will be allocated two words.

NAMES - EXERCISES

Indicate whether the following are valid or invalid names:

Name	Valid	Invalid
VELOCITY		
BETA1		
2BETA		
$$		
OUTOFSTOCK		
STOCK NUMBER		
STOCKNO.		
STAT'BIT		

ANSWERS

Indicate whether the following are valid or invalid names:

Name	Valid	Invalid
VELOCITY	X	
BETA1	X	
2BETA		X (begins with number)
$$	X	
OUTOFSTOCK	X	
STOCK NUMBER		X (blank not allowed)
STOCKNO.		X (period not allowed)
STAT'BIT	X	

Integer Item-Preset

An item-preset provides an initial value for an item declared to be a variable and a constant value for an item declared to be CONSTANT (discussed next). An integer literal (a sequence of one or more digits interpreted as a decimal value) may be used to preset integer items. An item preset follows the type-description:

ITEM name type-description **item-preset** ;

The item-preset looks like this:

= value

For example:

ITEM	name	type-description		item-preset	semi-colon	
↓	↓	↓		↓	↓	
ITEM	SPEED	U 10	=	967	;	SPEED is a 10 bit unsigned integer variable which is preset to 967.
ITEM	DISTANCE	S 6	=	33	;	DISTANCE is a 6 bit signed integer variable preset to 33.
ITEM	AREA	U	=	3	;	AREA is a single-word unsigned integer variable which defaults to the size of one word and is preset to 3.
ITEM	METERS	S	=	42	;	METERS is a single word signed integer variable which defaults to the size of one word and is preset to 42.

NOTES: An item-preset gives an initial value to an item. All items that are not preset have unknown values. That is, there is no implicit system initialization for items that are not preset.

It is the programmer's responsibility to ensure that the value of the item-preset can be represented in the explicit or default integer-size.

A negative value may be used to preset a signed integer:

ITEM N'METERS = -42;

The preset value in this case is syntactically an integer formula, not an integer literal.

<u>Constant Integer Items</u>

An item may be declared to be constant, in which case its value cannot be changed during execution. A constant item <u>must</u> be given an initial value by means of an item-preset. The form of a constant item - declaration is as follows:

CONSTANT item name type-description **item-preset** ;

<u>Examples</u>:

CONSTANT ITEM FIRSTNUM U 5 = 27;

CONSTANT ITEM LOWESTVAL S = 0;

CONSTANT ITEM DOLLARS U 10 = 1000;

CONSTANT ITEM $ALL S = 57;

A consistant-item always maintains its preset value, a constant-item may not have its value changed within a program.

INTEGER TYPE-DECLARATIONS

A type-declaration allows a programmer to declare a new, possibly more mnemonic name to describe the type of an item. The type-declaration sets up a template that may be used in place of a type-description. The form of an item type-declaration is:

TYPE item-type-name type-description;

The item type-declaration associates the item type-name with the attributes of the type-description.

> *NOTE: A type-declaration has no item-preset because it declares no data. A type-declaration creates a template. It defines what data described using a particular type-name will look like.*

Once a type-name has been declared, it may then be used in <u>any</u> item-declaration in place of the explicit type-description, as shown in the syntax below:

$$\text{[CONSTANT] ITEM name} \left\{ \begin{array}{l} \text{type-description} \\ \text{item-type-name} \end{array} \right\} \text{[item-preset];}$$

For example:

TYPE DIGITS S 7;

This type-declaration declares an item type-name DIGITS that may be used to describe the type of data objects. The declaration of DIGITS sets up a template for seven-bit signed integers, and may be used as follows:

ITEM MAXVAL DIGITS;

MAXVAL is declared to be a signed integer item, allocated seven bits of storage for its value and one sign bit.

Examples:

Having made the following type-declarations:

```
TYPE DIGITS S 7;

TYPE COUNT U 10;
```

The following are correct uses of the item-type-names DIGITS and COUNT in item-declarations:

	Explanation
ITEM SUM DIGITS = 17;	SUM is declared to be a 7 bit signed integer item preset to 17.
ITEM MAXNUM DIGITS;	MAXNUM is a seven bit signed integer item.
ITEM XCOORD COUNT;	XCOORD is declared to be of type COUNT, which makes it able to store unsigned integer values of up to 10 bits.
ITEM NUMBER COUNT = 0;	NUMBER is of type COUNT and is preset to zero.
ITEM SOME'VALUE COUNT;	SOME'VALUE is of type U 10
ITEM TOTAL$ DIGITS;	TOTAL$ is declared to be of type DIGITS, and can assume signed integer values which require 7 bits or less of storage.
CONSTANT ITEM AMOUNT COUNT = 49;	AMOUNT is a constant item of type COUNT, whose value will always be 49.
ITEM GUESS DIGITS;	GUESS is a 7 bit signed integer variable.

There are two advantages to declaring and using item-type-names instead of explicit type-descriptions. First, the item-type-name itself may provide some information to the reader about the intended use and class of an item. For example, if several items are to be used as counters, an item type-declaration could be written and used as follows:

```
TYPE COUNTER S 15;
ITEM OUTER'LOOP COUNTER;
ITEM MIDDLE'LOOP COUNTER;
ITEM INNER'LOOP COUNTER;
```

Second, using the item-type-name makes global changes quite simple.

INTEGER ITEM-DECLARATIONS - EXERCISES

Given the following type-declaration:

TYPE SHORTNUM S 10;

Determine whether the following item-declarations are correct or incorrect:

Declaration	Correct	Incorrect
1. ITEM NAME U 7 = 1;		
2. ITEM NO.NUM S = 13;		
3. ITEM PLANE SHORTNUM;		
4. ITEM AREA SHORTNUM = 2000;		
5. CONSTANT ITEM TRI'SIDE S 4;		
6. CONSTANT ITEM SQUARE U 8 = 7;		
7. CONSTANT ITEM CIRCLE SHORTNUM = 36;		
8. ITEM DISTANCE SHORTNUM S 10;		

ANSWERS

Given the following type-declaration:

TYPE SHORTNUM S 10;

Determine whether the following item-declarations are correct or incorrect:

Declaration	Correct	Incorrect
1. ITEM NAME U 7 = 1;	X	
2. ITEM NO.NUM S = 13;		X (period not allowed)
3. ITEM PLANE SHORTNUM;	X	
4. ITEM AREA SHORTNUM = 2000;	X (but code erroneous)	
5. CONSTANT ITEM TRI'SIDE S 4;		X (need preset)
6. CONSTANT ITEM SQUARE U 8 = 7;	X	
7. CONSTANT ITEM CIRCLE SHORTNUM = 36;	X	
8. ITEM DISTANCE SHORTNUM S 10;		X (either SHORTNUM or S 10, not both)

FLOATING POINT ITEM-DECLARATIONS

Another type of JOVIAL (J73) item is the floating point item, used to represent numbers with fractional values. The general syntax for floating point item-declarations is the same as shown for integers:

[CONSTANT] ITEM name $\left\{ \begin{matrix} \text{type-description} \\ \text{item-type-name} \end{matrix} \right\}$ [item-preset];

The item-declaration for a floating point item might look like this:

ITEM	name	type-description	semi-colon
↓	↓	↓	↓
ITEM	RADIUS	F 20	;

$\boxed{\text{ITEM}}$ JOVIAL (J73) reserved word.

$\boxed{\text{name}}$ A legal JOVIAL (J73) name (unique and supplied by the programmer)

$\boxed{\text{type-description}}$ The type-description for a floating point item consists of a single letter type-indicator and an optional precision specifier. The type-indicator is

F - floating point type

Precision specifies the minimum number of bits of storage to be allocated to hold the mantissa. (Precision does not include any exponent or sign bits.) If the precision is omitted, it defaults to FLOATPRECISION. (FLOATPRECISION is an implementation-parameter equal to the number of bits needed for a single-precision floating point representation.)

$\boxed{;}$ A semi-colon terminates the declaration.

The following item-declaration:

```
ITEM RADIUS F 20;
```

declares an item named RADIUS to be of floating point type, and allocated at least 20 bits for the mantissa of the floating point representation. The following are more examples of floating point item declarations:

ITEM	name	type-description	semi-colon	
↓	↓	↓	↓	
ITEM	AZIMUTH	F 30	;	AZIMUTH is a floating point variable allocated at least 30 bits for its mantissa.
ITEM	VELOCITY	F	;	VELOCITY is a floating point variable. If FLOATPRECISION is 23, it is allocated 23 bits for its manitssa.

> *NOTES: As with integers, in many cases a programmer can choose to ignore thenumber of bits allocated for a floating point item and use the compiler default. There are only two sizes of floating point types that are allocated: single precision or double precision. For example, if FLOATPRECISION is 23, the following items are allocated as single precision floating point items:*
>
> *ITEM RADIUS F 20;*
>
> *ITEM SEARCH F;*
>
> *ITEM LIMIT 5;*
>
> *The precision specifier in floating point type-description must be used to indicate a double precision floating point type:*
>
> *ITEM MEASURE F 39;*

Floating Point Item-Presets

As mentioned previously, an item preset provides an initial value for an item declared to be a variable and a constant value for an item declared to be constant. A real literal may be used to preset floating point data. A real literal is any one of the following:

Decimal number	12. 6.097
Decimal number followed by an exponent	436.123E5 2.97E-7
Integer followed by an exponent	256E-8 1E4

The form of a real literal with an exponent is interpreted as follows:

$$1E4 \rightarrow 1*10^{4} \rightarrow 1000$$

(The positive exponent causes the decimal point to be interpreted that many places to the right.)

As another example:

$$2.97E-3 \rightarrow 2.97*10^{-3} \rightarrow .00297$$

(The negative exponent causes the decimal point to be interpreted that many places to the left.)

The form of an item-preset is as follows:

ITEM name type-description $\boxed{\text{item-preset}}$;

where the preset looks like this:

= value

> *NOTE: The only difference between integer and floating point item-presets is that the latter are preset with real literals as opposed to integer literals.*

Examples:

ITEM	name	type-description		item-preset	semi-colon	
↓	↓	↓		↓	↓	
ITEM	RADIUS	F 20	=	3.912	;	RADIUS is a floating point variable allocated at least 20 bits of precision and preset to 3.912.
ITEM	SEARCH	F	=	1.6E3	;	Item SEARCH is a floating point variable allocated a default number of bits of precision (FLOATPRECISION) and preset to 1.6E3.
ITEM	LIMIT	F 30	=	16.	;	LIMIT is a floating point variable allocated at least 30 bits of precision and preset to 16.

> *NOTES: An item-preset gives an initial value to an item. All that are not preset have unknown value. That is, there is no implicit system initialization for items that are not preset. A negative value may be used to preset a floating point item:*
>
> *ITEM N-MEASURE F = -6.725;*
>
> *The preset value in this case is syntactically a floating formula, not a floating literal.*

Constant Floating Point Items

As with integers, floating point items may be declared to be constant. The value of a CONSTANT item cannot change during program execution. A constant item must be given an initial value by means of an item preset. The form of a CONSTANT item declaration is:

[CONSTANT] ITEM name type-description [item preset] ;

Examples:

```
CONSTANT ITEM ORIG'VAL F 10 = 16.29;
CONSTANT ITEM NEXT'VAL F = .0629E4;
CONSTANT ITEM COEFFICIENT F 20 = 21.36;
CONSTANT ITEM DISTANCE F = .001;
```

FLOATING POINT TYPE-DECLARATIONS

As mentioned in the section on integers, a type-declaration may be used to allow a programmer to declare a new name to describe the type of an item. The type-declaration sets up a template that may be used in place of a type-description. The form of an item type-declaration is:

TYPE item-type-name type-description;

The item type-declaration associates the item-type-name with the attribute of the explicit type-description.

> *NOTE: A type-declaration has no item-preset because it declares no data. A type-declaration creates a template. It defines what data described using a particular type-name will look like.*

Once an item-type-name has been declared, it may be used in any item-declaration in place of the explicit type-description, as shown in the syntax below:

[CONSTANT] ITEM name $\left\{\begin{array}{l}\text{type-description}\\\text{item-type-name}\end{array}\right\}$ [item-preset];

For example,

TYPE LONG'FLOAT F 39;

This type-declaration declares an item-type-name LONG'FLOAT that may be used to describe the type of data objects. The declaration of LONG'FLOAT sets up a template for floating point types with at least 39 bits of precision that may be used as follows:

ITEM MEASURE LONG'FLOAT;

MEASURE is declared to be a floating point type with at least 39 bits of precision by using the item-type-name LONG'FLOAT instead of the explicit type-description.

Examples:

Having made the following type-declaration:

```
TYPE LENGTH F 12;
```

The following are correct uses of the item-type-name LENGTH in an item-declaration:

```
ITEM VARX LENGTH;
ITEM TAXES LENGTH = .162E2;
CONSTANT ITEM PI LENGTH = 3.14159;
```

FLOATING POINT ITEM-DECLARATIONS - EXERCISES

Determine whether the following item declarations are correct or incorrect.

Given the following type-declaration:

TYPE LONGNUM F 30;

Declaration		Correct	Incorrect
1.	ITEM XCOORD F 20;		
2.	ITEM PEOPLE LONGNUM = 3.629;		
3.	CONSTANT ITEM NAMES;		
4.	CONSTANT ITEM BOOKS'IN'PRINT = 3E-67		
5.	ITEM RIGHT LONGNUM = .000001;		
6.	ITEM NO CASE LONGNUM;		
7.	ITEM AA F 15 = 69;		

ANSWERS

Determine whether the following item declarations are correct or incorrect.

Given the following type-declaration:

```
TYPE LONGNUM F 30;
```

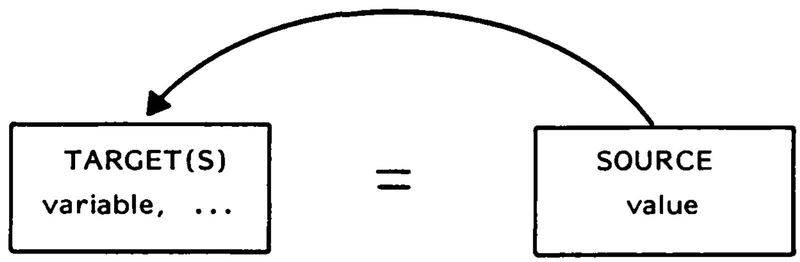
	Declaration	Correct	Incorrect
1.	ITEM XCOORD F 20;	X	
2.	ITEM PEOPLE LONGNUM = 3.629;	X	
3.	CONSTANT ITEM NAMES;		X (need type, preset)
4.	CONSTANT ITEM BOOKS'IN'PRINT = 3E-67		X (need type, ;)
5.	ITEM RIGHT LONGNUM = .000001;	X	
6.	ITEM NO CASE LONGNUM;		X (blank in name)
7.	ITEM AA F 15 = 69;		X (need real preset)

SECTION 4

ASSIGNMENT-STATEMENTS

ASSIGNMENT-STATEMENTS

One important use of integer and floating point data items is in assignment-statements. An assignment-statement causes the value to the right of the equal sign (the source) to become the value of the variable or variables to the left of the equal sign (the target):



A simple assignment-statement assigns the value of a source to be the value of one target variable. For example:

```
DISTANCE = 2.5;

CASH = 40;

DENSITY = ORIG'VAL;

SCAN = 3.2E-7;
```

A multiple assignment-statement assigns a source value to more than one target variable. For example:

```
LENGTH, WIDTH, HEIGHT = 0;
```

> *NOTE: In performing the assignment, the formula (or source) is evaluated first. Then, the leftmost variable is evaluated and the value of the formula is assigned to that variable. Next, the second-to-the-left variable is evaluated and the value of the formula is assigned to it. This sequence of evaluations continues until the list of variables is exhausted.*

For an assignment-statement to be correct semantically, the type of the source value must be equivalent or implicitly convertible to the type of the target variable(s) receiving the assignment.

Equivalent types are those that the compiler recognizes as being the same. Neither the programmer nor the compiler needs to provide any additional information to make the types match. Two integer types are eqivalent if they are both unsigned or both signed, and they have the same integer-size (explicitly or by default). Two floating point types are equivalent if they have the same precision (explicitly or by default).

Implicitly convertible types are those types that the compiler recognizes as being almost the same. The programmer doesn't have to provide any additional information to the compiler, but the compiler automatically generates some code to make the types match.

Any integer type is implicitly convertible by the compiler to any other integer type. Only smaller precision floating point types are implicitly convertible by the compiler to a larger floating point type.

Look at sample integer types and how the type matching rules apply to assignment-statements.

```
TYPE SMALL S 10;

ITEM ROOM S 10;

ITEM CARS U 6;

ITEM MAGAZINES SMALL;
```

MAGAZINES and ROOM have EQUIVALENT types - they are both S 10's - so the following assignment-statements may be made:

```
MAGAZINES = ROOM;

ROOM = MAGAZINES;
```

The type of CARS is IMPLICITLY CONVERTIBLE to the type of MAGAZINES - they are both integer types. The following assignment can be made:

```
MAGAZINES = CARS;
```

The type of MAGAZINES is sufficient for all possible run time values of CARS.

The type of MAGAZINES is IMPLICITLY CONVERTIBLE to the type of CARS - they are both integer types.

The following is a legal assignment:

CARS = MAGAZINES;

However, the type of cars is not sufficient for all possible run-time values of MAGAZINES. If the run-time value of MAGAZINES is negative or greater than 63, the results of the assignment are unpredictable.

Given the following declarations:

TYPE BALANCE S 6;

TYPE PROFIT U 10;

ITEM TOTAL S 6;

ITEM CLIMAX U 4;

ITEM SAVINGS S 10;

ITEM CHECK BALANCE;

ITEM BANK PROFIT;

The following assignment-statements have the following results:

TOTAL = CHECK;	The types are equivalent; no conversion is needed; the assignment is made.
BANK = CLIMAX;	The type of CLIMAX is implicitly converted to the type of BANK, and the assignment is made.
CLIMAX = BANK;	The assignment is made; if the run-time value in BANK cannot fit in CLIMAX, the results are unpredictable.
SAVINGS = TOTAL;	The type of TOTAL is implicitly converted to the type of SAVINGS, and the assignment is made.
CHECK = SAVINGS;	The assignment is made; if the run-time value in SAVINGS can not fit in CHECK, the results are unpredictable.
BANK = CHECK;	The assignment is made, if the run-time value in CHECK can not fit in BANK, the results are unpredictable.

Now look at floating point types, and how the type matching rules apply.

Given the following declarations:

```
TYPE TAX'DUE F 20;

ITEM I'O'IRS TAX'DUE;

ITEM IRS'OWES'ME F 20;

ITEM FIGURE'TAX F 30;
```

I'O'IRS and IRS'OWES'ME have EQUIVALENT types - they are both F 20. The following assignment-statements can be made:

```
I'O'IRS  = IRS'OWES'ME;

IRS'OWES'ME = I'O'IRS;
```

The type of IRS'OWES'ME is IMPLICITLY CONVERTIBLE to the type of FIGURE'TAX. The precision of FIGURE'TAX is greater than that of IRS' OWES'ME. The following is a legal assignment:

```
FIGURE'TAX = IRS'OWES'ME;
```

The type of FIGURE'TAX is NOT implicitly convertible to the type of either I'O'IRS or IRS'OWES'ME since the precision of FIGURE'TAX is greater than that of either I'O'IRS or IRS'OWES'ME. The value of FIGURE'TAX may not be assigned to I'O'IRS or IRS'OWES'ME.

Again, for an assignment-statement to be correct, the type of the source value must be equivalent or implicitly convertible to the type of the target variable.

Examples:

Given the following declarations:

```
TYPE SHORT'FLOAT F 5;

TYPE LONG'INT U 15;

ITEM NAME U 10;

ITEM DATE S 4;
```

ITEM TOTAL LONG'INT;

ITEM INDEX SHORT'FLOAT;

ITEM DIMENSION F 20;

The following are illegal assignment-statements:

TOTAL = DIMENSION; DIMENSION is a floating point type; TOTAL is an integer type. There is no implicit conversion defined from float to integer; the assignment is incorrect.

INDEX = DIMENSION; DIMENSION is a long floating point type; INDEX is a short floating point type. There is no implicit conversion defined from long float to short float; the assignment is incorrect.

DIMENSION = DATE; DATE is an integer type; DIMENSION is a floating point type. There is no implicit conversion defined from integer to float; the assignment is incorrect.

The following are legal assignment-statements:

DIMENSION = INDEX; Implicit conversion is defined from short float to long float.

NAME = DATE;

DATE = TOTAL;

TOTAL = DATE;

Implicit conversion IS defined from any integer type to any other integer type. (If the run-time value of the source is out of range of the target, the results of the assignment are unpredictable.)

Explicit Conversion

An explicit conversion may be used to temporarily impose a template of one type on a value of another type. If the source in an assignment-statement is not equivalent or implicitly convertible to the type of the target(s) receiving the assignment, a conversion-operator may be used on the source value to make the types match.

For instance, given the following item-declarations:

ITEM DISTANCE S 10;

ITEM SPEED F 20;

The assignment of SPEED to DISTANCE:

```
DISTANCE = SPEED;
```

is not allowed; the type of SPEED (floating point) is not equivalent or implicitly convertible to the type of DISTANCE (integer). A conversion-operator may be used to temporarily place a 10-bit signed integer template on the value of SPEED so the assignment may be made:

```
                  conversion-operator   value
                         ↓                ↓
DISTANCE   =      (*S 10*)              (SPEED) ;
```

A conversion operator is of the form

```
conversion-operator (value)
```

A conversion-operator can be any of the following:

Operator	Example
(* type-description *)	(* U 5 *)
type-indicator	U, S, F
item-type-name	COUNTER (where TYPE COUNTER U 10;)
(* item-type-name *)	(* COUNTER *)

> *NOTE: It is preferred programming practice to enclose conversion operators in parentheses and asterisks. This makes it more apparent to the reader that a conversion is taking place.*

Explicit Conversion - Examples

Declarations

```
TYPE COUNTER U 10;

TYPE SUMMIT F 10;

ITEM PAGE S 5;

ITEM BOOK F 8;

ITEM NAME U 10;

ITEM MINUTES F 15;

ITEM TIME SUMMIT;
```

Consider the following conversions:

1.	(* U 10 *) (BOOK)	U(BOOK)	COUNTER(BOOK)
2.	(* F 8 *) (MINUTES)		
3.	(* SUMMIT *) (PAGE)	SUMMIT(PAGE)	F(PAGE)
4.	(* U 10 *) (PAGE)		
5.	SUMMIT(MINUTES)	(* SUMMIT *)(MINUTES)	

Explanation

1. The floating point value of BOOK is placed under three integer templates. The first conversion-operator is the parentheses-asterisk around the type-description. BOOK is considered as a 10-bit unsigned integer. The second conversion-operator is a single letter type-indicator, U. BOOK is considered as a single-word unsigned integer. The third conversion operator is the item-type-name COUNTER.BOOK is considered to be of type COUNTER, a 10-bit unsigned integer.

2. MINUTES, a long floating point item is placed under a shorter floating point template. The parentheses-asterisk around the type-description is used as the conversion-operator.

3. The integer value of PAGE is placed under three floating point templates. The first conversion-operator is the parentheses-asterisk around the item-type-name. PAGE is considered to be of type SUMMIT, a floating point type with at least 10 bits of precision. The second conversion-operator is simply the item-type-name. This explicit conversion has exactly the same effect as the previous explicit conversion. However, by using the parentheses-asterisk form, it is asserted to the reader that an explicit conversion is occurring. The third conversion-operator is a single letter type-indicator, F. PAGE is considered as a single precision floating point type.

4. While it is true that any integer type is implicitly convertible to any other integer type, a piece of code may be clearer by including an explicit conversion to assert to the reader that the programmer is knowingly doing such a conversion. The conversion-operator is the parentheses-asterisk around the type-description. PAGE is considered as a 10-bit unsigned integer.

5. MINUTES, a longer floating point item, is considered to be of type SUMMIT, a shorter floating point type, in both conversions. The first conversion-operator is item-type-name SUMMIT; the second conversion-operator is the parentheses-asterisk around the item-type-name.

The different forms of the conversion-operator may be used on the source in an assignment-statement to comply with the type matching rules of assignment.

Declarations

```
TYPE COUNTER U 10;

TYPE SUMMIT F 10;

ITEM PAGE S 5;

ITEM BOOK F 8;

ITEM NAME U 10;

ITEM MINUTES F 15;

ITEM TIME F 10;
```

Incorrect Syntax (The type of the source is not implicitly convertible to the type of the target.)		Correct Syntax (These are not the "only" correct answers.)
1.	NAME = BOOK;	NAME = (* U 10 *) (BOOK);
2.	BOOK = MINUTES;	BOOK = (* F 8 *) (MINUTES);
3.	MINUTES = PAGE;	MINUTES = (* F 10 *) (PAGE);
4.	NAME = BOOK;	NAME = U(BOOK);
5.	TIME = MINUTES;	TIME = SUMMIT(MINUTES);

Explanation

1. Since a floating point type is not implicitly convertible to an unsigned integer type, a conversion-operator must be used. BOOK is considered as a 10-bit unsigned integer, the type of NAME, and the assignment is made.

2. Since a longer floating point type is not implicitly convertible to a shorter floating point type, a conversion-operator must be used. MINUTES is considered as a shorter floating point type, the type of BOOK, and the assignment is made.

3. Since a signed integer type is not implicitly convertible to a floating point type, a conversion-operator must be used. PAGE is considered as an F 10, which is then implicitly converted to the type of MINUTES, F 15, and the assignment is made.

4. Since a floating point type is not implicitly convertible to an unsigned integer type, a conversion-operator must be used. BOOK is considered as a single-word unsigned integer, which is then implicitly converted to the type of NAME, U 10, and the assignment is made.

5. Since a longer floating point type is not implicitly convertible to a shorter floating point type, a conversion-operator must be used. MINUTES is considered to be of type SUMMIT, an F 10, the type of TIME, and the assignment is made.

The Round-or-Truncate Attribute

The round-or-truncate attribute may be used to further control a conversion. It may be given following the single letter type-indicator in a type-description:

U [, round-or-truncate]

S [, round-or-truncate]

F [, round-or-truncate]

R indicates rounding, T indicates truncation toward minus infinity.

> *NOTE:*
>
> - *If a round-or-truncate attribute is omitted, truncation occurs in a machine-dependent manner, either toward zero or toward minus infinity.*
> - *The round-or-truncate attribute does not follow the single letter type-indicator when used by itself as a conversion-operator.*
>
> *Examples of incorrect use:*
>
> *S, T (BB)*
>
> *F, R (BB)*

Given the following declarations:

```
TYPE SHORTNUM S,T 5;

TYPE LENGTH F,R 10;

ITEM BB F 15;
```

The following conversions may be made correctly:

(*SHORTNUM*)(BB) — The type of item BB will be converted to S 5, with truncation of any fraction occurring toward minus infinity.

(* S,T 5 *) (BB)

(*LENGTH*)(BB)	The type of item BB will be converted to F 10, with rounding.
(* F,R 10 *)(BB)	

Examples

(* S,R *) (3.5) → 4	
(* S,T *) (3.5) → 3	
(* S *) (3.5) → 3	(if machine-dependent truncation is either toward zero or toward minus infinity)
(* S,R *) (-3.5) → -3	
(* S,T *) (-3.5) → -4	
(* S *) (-3.5) → -4	(if machine-dependent truncation is toward minus infinity)
(* S *) (-3.5) → -3	(if machine-dependent truncation is toward zero)

SECTION 5

FORMULAE

SOFTECH

FORMULAE

A formula is either a single operand or a combination of operators and operands. A formula may be parenthesized and the result of a formula has a type. This section discusses integer and floating point formulae.

Operator Precedence

The order in which operators and operands are combined is determined by the precedence of the operators. Operators of equal precedence are evaluated from left to right, if the operation is not commutative (i.e., in addition, the order of evaluation is determined by the most efficient code which can be generated since 5 + 3 is the same as 3 + 5). Expressions within parentheses are evaluated first, from the inner most parenthesis out.

A complete set of JOVIAL (J73) operators and their precedence is as follows:

Precedence	Operator
6	@, subscripting, function calls
5	**
4	*, /, MOD
3	+, -
2	=, <>, <, >, <=, >=
1	NOT, AND, OR, EQV, XOR
0	assignment

> *NOTE: Many of these operators will be discussed in later workbooks. They are listed here to provide an overview.*

Integer Formulae

An integer formula has integer operands (signed and/or unsigned of any length), and returns an integer result. Integer formulae may use the following operators:

**

*, /, MOD

+, -

> *NOTES: The quotient in integer division is computed exactly and truncated, in an implementation-dependent way, to an integer result. The divisor (the second operand) must not be zero. The MOD operator returns the integer remainder of the integer division. For example:*
>
> *7 MOD 3 → 1*
>
> *Seven divided by three is two, remainder one; the result of the MOD is the integer remainder. The divisor (the second operand) must not be zero.*

Integer exponentiation is done only if the second operand, the exponent, is a non-negative integer known at compile-time. Since integer exponentiation is done as repeated multiplication, the compiler must know how many times it is to do the multiplication.

Examples:

Declarations

```
ITEM FIRST'U;

ITEM SECOND'S;
```

Formulae

```
FIRST + SECOND'

FIRST - SECOND'

FIRST * SECOND'
```

FIRST / SECOND'

FIRST ** 2

FIRST MOD SECOND'

FIRST + (SECOND' + FIRST')

<u>The Type of an Integer Formula</u>

The result of an integer operation has the type:

S n

where n is one less than the multiple of BITSINWORD needed to hold the larger operand.

In other words, if the operands of an integer formula are single-word integers, the type of the result of an integer formula is a single-word signed integer. If any operand is a double-word integer, the type of the result of an integer-formula is a double-word signed integer.

<u>Declarations</u>

ITEM FIRST' S 10 = 14;

ITEM SECOND' U 30 = 10;

Examples	Result	Type (BITSINWORD=16)	Type (BITSINWORD=36)
FIRST' + SECOND'	24	S 31	S 35
FIRST' - SECOND'	4	S 31	S 35
FIRST' * SECOND'	140	S 31	S 35
FIRST' / SECOND'	1	S 31	S 35
FIRST' ** 2	196	S 15	S 35
FIRST' + FIRST'	28	S 15	S 35
FIRST' MOD SECOND'	4	S 31	S 35
FIRST' + (SECOND' + FIRST')	38	S 31	S 35

If BITSINWORD is 16, FIRST' is allocated one word and SECOND' is allocated two words. Any formula referencing SECOND' has a double-word integer for one of its operands, so the type of the result for those formulae are double-word signed integers. Any formula referencing only FIRST' or using an integer literal that can be represented in only a single-word has only single-word integer operands, so the type of the result for those formulae are single-word signed integers.

If BITSINWORD is 36, both FIRST' and SECOND' are allocated one word. The type of the result for any formula using them is a single-word signed integer.

OPERATOR PRECEDENCE - EXERCISES

Evaluate the following formulae and give the resulting value and type.

Declarations

```
TYPE LONGNUM U 20;
ITEM LENGTH LONGNUM = 20;
ITEM WIDTH LONGNUM = 10;
ITEM HEIGHT S 7 = 5;
```

Formulae	Value	Type (BITSINWORD=16)
1. LENGTH + WIDTH / HEIGHT		
2. LENGTH + HEIGHT ** 2 / WIDTH		
3. (LENGTH MOD WIDTH) * HEIGHT		
4. WIDTH / HEIGHT ** 2		
5. 2 ** 3 ** 2		
6. LENGTH * WIDTH / HEIGHT		

ANSWERS

Evaluate the following formulae and give the resulting value and type.

Declarations

```
TYPE LONGNUM U 20;
ITEM LENGTH LONGNUM = 20;
ITEM WIDTH LONGNUM = 10;
ITEM HEIGHT S 7 = 5;
```

Formulae	Value	Type (BITSINWORD=16)
1. LENGTH + WIDTH / HEIGHT	22	S 31
2. LENGTH + HEIGHT ** 2 / WIDTH	22	S 31
3. (LENGTH MOD WIDTH) * HEIGHT	0	S 31
4. WIDTH / HEIGHT ** 2	0	S 31
5. 2 ** 3 ** 2	64	S 15
6. LENGTH * WIDTH / HEIGHT	40	S 31

ITEM-DECLARATIONS - INTEGERS - SUMMARY EXERCISES

Assume BITSINWORD = 16.

1. **Declare an integer item capable of taking on values -37 through +20.**

2. **Declare an integer item always to be equal to your present age.**

3. **Declare an integer item that will be used to measure your weight.**

4. **Write a formula to cube your present age, subtract off your weight, find the remainder when divided by the item declared in # 1. What is the type of the result?**

5. **Declare an integer item and assign to it the value of the formula in # 4.**

ANSWERS

1.
```
ITEM VALUE S;
ITEM VALUE S 6;
```

2.
```
CONSTANT ITEM AGE U = 27;
CONSTANT ITEM AGE U 5 = 27;
```

3.
```
ITEM WEIGHT U;
ITEM WEIGHT U 7;
```

4.
```
(AGE ** 3 - WEIGHT) MOD VALUE
S 15
```

5.
```
ITEM ANSWER S 15;
ANSWER = (AGE ** 3 - WEIGHT) MOD VALUE;
```

FLOATING POINT FORMULAE

A floating point formula has floating point operands or is a floating point exponentiation formula, and returns a floating point result. Floating point formulae may use the following operators:

**

*, /

+, -

NOTES:

- *The divisor (the second operand) in division must not be zero.*
- *The MOD operator is undefined for floating point formulae.*
- *The precision attribute of a floating-formula is that of the formula's most precise operand. The operand of a floating-conversion is first computed according to the default rules, and then converted to the specified floating type.*
- *A floating-point exponentiation formula handles all exponentiation that cannot be done as integer exponentiation. The type of the operands of floating exponentiation may both be floating point one integer and one floating point, or both integer (where the second operand - the exponent - is negative, not known at compile time or both). Floating point exponentiation is computed using logarithms, so the first operand, the base of the exponentiation formula, must not be negative.*

Examples:

Declarations

```
ITEM FIRST' F;

ITEM SECOND' F;
```

Examples

```
FIRST' + SECOND'

FIRST' - SECOND'

FIRST' * SECOND'

FIRST' / SECOND'

FIRST ' ** SECOND'

FIRST' * ( SECOND' * FIRST' )
```

The Type of a Floating Point Formula

The result of a floating point operation has the type:

F p

where p is the precision of the larger operand.

Declarations

```
ITEM FIRST F 10 = 5.6;

ITEM SECOND F 15 = .8;
```

Examples	Result	Type
FIRST' + SECOND'	4.8	F 15
FIRST' - SECOND'	6.4	F 15
FIRST' * SECOND'	-4.48	F 15
FIRST' / SECOND'	-7.	F 15
FIRST' ** SECOND'	$5.6^{(-.8)}$	F 15
FIRST' * (SECOND' * FIRST')	-25.088	F 15
FIRST' + FIRST'	11.2	F 10

Any formula referencing SECOND' has the result type of F 15. Any formula referencing only FIRST' has a result of F 10.

OPERATOR PRECEDENCE - EXERCISES

Evaluate the following formulae and give the resulting value and type.

<u>Declarations</u>

```
TYPE SHORTNUM F 8;
ITEM LENGTH SHORTNUM = 3.6;
ITEM WIDTH SHORTNUM = -.02;
ITEM HEIGHT F 20 = 1.1;
```

Formulae	Value	Type
1. LENGTH + HEIGHT / WIDTH		
2. WIDTH + HEIGHT ** 2		
3. HEIGHT - WIDTH + LENGTH		
4. HEIGHT - (WIDTH + LENGTH)		

ANSWERS

Evaluate the following formulae and give the resulting value and type.

Declarations

```
TYPE SHORTNUM F 8;
ITEM LENGTH SHORTNUM = 3.6;
ITEM WIDTH SHORTNUM = -.02;
ITEM HEIGHT F 20 = 1.1;
```

Formulae	Value	Type
1. LENGTH + HEIGHT / WIDTH	-51.4	F 20
2. WIDTH + HEIGHT ** 2	1.19	F 20
3. HEIGHT - WIDTH + LENGTH	4.72	F 20
4. HEIGHT - (WIDTH + LENGTH)	-2.48	F 20

SECTION 6

SUMMARY

SOFTECH

SUMMARY

This first workbook covered the general organization of a program and the specifics of two types of item-declarations.

The general format of JOVIAL (J73) program is:

```
START
PROGRAM name;
      BEGIN
      "declarations"
      "executable statements"
      "subroutines"
      END
TERM
```

The START-TERM pair delimit the module (the compilation unit). The program is given a name, and a BEGIN-END pair surround the program-body. All data objects used in a program must be declared. The executable statements define and control the algorithm of the complete program. Portions of that program may be factored out and written as small programs themselves in subroutines.

A general form of an item-declaration is:

$$\text{[CONSTANT] ITEM name} \begin{Bmatrix} \text{type-description} \\ \text{item-type-name} \end{Bmatrix} \text{[item-preset];}$$

Name must be at least two characters long, beginning with a letter or a dollar sign, containing any number of letters, dollar signs, digits, or primes (single quote characters).

Integer and floating point type-descriptions were discussed in this section. The forms of those type-descriptions are:

U [integer-size]	for non-negative whole number values
S [integer-size]	for negative and non-negative whole number values
F [precision]	for values with fractional portions

If integer-size is not given, the compiler supplies a default, BITSINWORD-1 (a single word integer).

If precision is not given, the compiler supplies a default, FLOATPRECISION (a single precision floating point).

An item-type-name may be declared to take on the attributes of an explicit type-description. Once declared, it may be used in place of the type-description in an item-declaration. The form of the item type-declaration is:

TYPE item-type-name type-description;

All items have unknown initial value unless they are preset in the item-declaration. Any literal value or any formula that is known at compile-time may be used as an item-preset. The item-preset is given following the type of the item, and it has the form:

= value

An item may be declared to be a constant by beginning the declaration with the word CONSTANT and presetting the item to its constant value. Constants are known at compile-time. They maintain their initial value throughout the scope of the entire program.

The value of an item or a formula may be assigned to be the value of another in an assignment-statement. The form is:

variable, ... = value;

The language further specifies that the type of the source value must be equivalent or implicitly convertible to the type of each target variable receiving an assignment. Equivalent types are those that the compiler recognizes as identical; neither the compiler nor the programmer has to provide any information to make the types match. Implicitly convertible types are those that the compiler recognizes as being slightly different, but it can automatically generate code to make the types match without any further information from the programmer. Any of the numeric types shown so far may be explicitly converted to any other numeric type. An explicit conversion is supplied by the programmer to inform the compiler about the type of a given formula.

The type matching rules are summarized in the chart below using the context of an assignment-statement.

Target	=	Source	
	equivalent	implicitly convertible	explicitly convertible
U	U same integer-size	U, S any integer-size	U, S any integer-size
S	S same integer-size	U, S any integer-size	U, S any integer-size
F	F same precision	F smaller precision	F any precision
			U, S any integer-size
			F any precision

The form of an explicit conversion is:

conversion-operator (formula)

The forms of the conversion-operator are:

(* type-description *)

type-indicator

item-type-name

(* item-type-name *)

The round-or-truncate attribute may be given in a type-description to further control conversion.

Integer and floating point types have a set of operators that they may use in formulae. The result of any formula has a type. The operators, the type of the operands they may use, and the type of the result of the formula are summarized in the chart below.

Formulae

Operator	Meaning	Type of Operands	Result Type
+	addition	S, U	S n, where n is the multiple of BITSINWORD to hold the larger operand minus 1
		F	F p, where p is the precision of the larger operand
-	subtraction	S, U	same as +
		F	same as +
*	multiplication	S, U	same as +
		F	same as +
/	division	S, U	same as +, result is truncated towards minus infinity
		F	same as +
**	exponentiation	S, U	same as +, exponent must be non-negative, compile-time known
		F, S, U	same as +, all exponentiation not handled by integer exponentiation
MOD	modulo	S, U	same as +, returns integer remainder of integer division

ITEM-DECLARATIONS - INTEGER AND FLOATING POINT -- REVIEW EXERCISES

Declarations

```
TYPE MAXVAL U, R 14;
TYPE MINVAL F, T 9;
ITEM AA U 6;
ITEM BB S 10;
ITEM CC F 9;
ITEM DD F 25;
ITEM EE F 10;
ITEM FF U 14;
```

Are the following formulae and assignment-statements correct or incorrect? If incorrect, use a conversion-operator to make the appropriate corrections.

Statement	C	I	Correct Statement
1. AA = BB / 6 ** 4;			
2. BB = EE MOD AA;			
3. CC = DD			
4. EE = AA * CC - BB;			
5. DD = EE ** 6.3 - CC / DD;			
6. FF = AA ** EE MOD DD;			

ANSWERS

Declarations

```
TYPE MAXVAL U, R 14;
TYPE MINVAL F, T 9;
ITEM AA U 6;
ITEM BB S 10;
ITEM CC F 9;
ITEM DD F 25;
ITEM EE F 10;
ITEM FF U 14;
```

Are the following formulae and assignment-statements correct or incorrect? If incorrect, use a conversion-operator to make the appropriate corrections.

Statement	C	I	Correct Statement
1. AA = BB / 6 ** 4;	X (code may be erroneous)		
2. BB = EE MOD AA;		X	BB = (*U*)(EE) MOD AA;
3. CC = DD		X	CC = (*F 9*)(DD);
4. EE = AA * CC - BB;		X	EE = (*F 10*)(AA) * CC - (*F 10*)(BB);
5. DD = EE ** 6.3 - CC / DD;	X		
6. FF = AA ** EE MOD DD;		X	FF = (*U*)(AA** (EE)) MOD (*U)

THE JOVIAL (J73) WORKBOOK

VOLUME 2

ITEM-DECLARATIONS, PART 2

1081-1

April 1981

Submitted to

Department of the Air Force
Rome Air Development Center
I S I S
Griffiss Air Force Base, NY 13441

Prepared by

SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02154

PREFACE

This workbook is intended for use with Tape 2 of the JOVIAL (J73) Video Course. Its purpose is to elaborate upon and reinforce concepts and language rules introduced in the videotape.

Fixed point, bit, character and status item types are addressed in this workbook. Each is discussed in terms of *general syntax*, type matching and where applicable, formulaic expressions. The final pages are a summary of information presented in this segment.

TABLE OF CONTENTS

SYNTAX

The syntax conventions used in the JOVIAL (J73) Video Course and workbook are as follows:

Syntax	Meaning	Sample Expansions
[some-feature]	Brackets indicate an optional feature.	some-feature OR nothing
{one | other}	Braces with a vertical bar indicate disjunction-a choice between alternatives.	one OR other
{this-one that-one}	Braces with each feature on separate lines indicate disjunction - a choice between alternatives.	this-one OR that-one
letter ...	The sequence '...' indicates one or more repetitions of a feature.	letter letter letter letter letter letter ...
(letter),...	The sequence "..." following a comma (or a colon) indicates one or more repetitions of the feature separated by commas (or colons).	(letter) (letter) (letter) (letter) (letter) (letter) ...
[{this-one that-one}] + another	Syntax symbols may be combined.	this-one + (another) that-one + (another) + (another)

SECTION 1

DATA-DECLARATIONS:
FIXED POINT ITEMS

DATA DECLARATIONS: FIXED POINT ITEMS

Fixed values are numbers with constant scale factors. They may be used to represent physical quantities (primarily to save time and/or storage) when the range of the value is narrow and predictable. For example, fixed values may be used in a computation that runs on a computer for which floating point hardware is not available or too slow.

The general syntax for item-declarations discussed in Workbook 1 holds true for fixed point items as well:

$$[\text{CONSTANT}]\ \ \text{ITEM}\ \begin{Bmatrix} \text{type-description} \\ \text{item-type-name} \end{Bmatrix}\ \ [\text{item-preset}];$$

ITEM ITEM is a JOVIAL (J73) reserved word. It has a specific meaning to the compiler, which restricts its use to declarations. It cannot be used as a name.

name A name is a sequence of letters, digits, dollar signs and primes (single quote characters). It must begin with a letter or a dollar sign, and be at least two characters long. A name may be any length, though only the first 31 characters are examined for uniqueness within the program. (Fewer than 31 characters may be examined for external names. External names are discussed in Workbook 8). [1]

type-description The type-description for fixed point items consist of the letter A, a scale and an optional fraction preceeded by a comma:

A scale [,fraction]

A scale must always be specified. If a fraction is not specified, the default size is the implementation parameter FIXED PRECISION minus the number of bits allocated for the scale.

The scale or fraction may be negative, but their sum must be positive. There are three possibilities:

---

[1]See list of JOVIAL (J73) reserved words given in Workbook 1, p. 1:3-2.

Scale Positive and Fraction Positive

The scale gives the number of bits to the left of the binary point and fraction gives the number of bits to the right of the binary point.

Example:

```
ITEM SUBTOTAL A 5, 2;
```

Interpretation:

```
SXXXXX.XX
```

(S represents the sign bit and X represents a bit of allocated storage.)

Scale Negative and Fraction Positive

If the scale is negative, the binary point is assumed to be that many bits to the left of the first non-sign bit of the representation.

Example:

```
ITEM INCREMENT A -10, 14;
```

Interpretation:

```
S.0000000000XXXX
```

(The bits filled with zeros are not allocated.)

Scale Positive and Fraction Negative

If the fraction is negative, the binary point is assumed to be that many bits to the right of the last bit of representation.

Example:

```
ITEM SIZE A 6, 5;
```

Interpretation:

```
SX00000.
```

$\boxed{;}$ A semi-colon terminates the declaration.

FIXED POINT ITEM-PRESETS

Optionally, a fixed point item can be **preset**. A **real literal** is used to preset a fixed point item. A real literal is one of the following:

Decimal number	12. 6.097
Decimal number followed by an exponent	436.123E5 2.97E-7
Integer followed by an exponent	256E-8 1E4

NOTE: A real literal is interpreted as a fixed point value when used in a fixed point context. (A real literal is said to be of ambiguous type - i.e., if it appears in a floating point context it is inferred to be a floating point type. If it appears in a fixed context, it is inferred to be of fixed point type.)

As with all item-declaration, the form of a preset is

= value;

CONSTANT FIXED POINT ITEMS

The programmer may also declare a fixed point item to be CONSTANT, which prevents its value from being changed during the execution of the program. The form is:

CONSTANT ITEM name $\left\{ \begin{array}{l} \text{type-description} \\ \text{item-type-name} \end{array} \right\}$ **item-preset;**

The reserved word CONSTANT and an item preset are required in addition to the constituents discussed above.

FIXED POINT ITEM-DECLARATIONS: EXAMPLES

ITEM MONEY A 20, 7;	MONEY is declared to be a fixed point item, allocated at least 20 bits for the value of its scale, and seven for its fraction.
CONSTANT ITEM RIGHT A 5 = .19E-7	RIGHT is a constant item whose value is .19E-7. The scale is at least 5 bits, and the fraction defaults to FIXEDPRECISION - the scale.
ITEM FIND'VAL A -2, 3;	FIND'VAL is declared to be a fixed point item allocated one bit of storage (fraction - scale). The binary point is assumed to be two places to the left of the first bit of representation, so the only values FIND'VAL can assume are zero or 1/8 (.125).
ITEM FOO A 4, -3 - 15.;	Item FOO is a fixed point item whose scale is 4 and whose binary point is assumed to be three places to the right of the first non-sign bit of representation. Since only one bit (the 4th significant bit) is allocated, FOO can assume values of zero or $2^4 - 1$ (15). Here it is preset to 15.

FIXED POINT TYPE-DECLARATIONS

As with integer and floating point items (discussed in Workbook 1), a fixed point template may be declared to describe the type of one or more items. The syntax of the item-type-declaration is as follows:

```
TYPE item-type-name type-description;
```

The item type-declaration associates the item type-name with the attributes of the type-description.

> *NOTE: A type-declaration has no item-preset because it declares no data. A type-declaration creates a template. It defines what data described using a particular type-name will look like.*

Once a type-name has been declared, it may then be used in <u>any</u> item declaration in place of an explicit type-description, as shown in the syntax below:

$$[\text{CONSTANT}]\ \ \text{ITEM}\ \ \text{name}\ \begin{Bmatrix}\text{type-description}\\ \textbf{item-type-name}\end{Bmatrix}\ [\text{item-preset}];$$

For example:

```
TYPE RANGE A 5, 2;
```

This type-declaration declares an item type-name RANGE which may be used to declare fixed point items of scale 5 and fraction 2. RANGE may be used as follows:

```
ITEM DISTANCE RANGE;
```

DISTANCE is declared to be a fixed point item with a scale of 5 and fraction of 2. (The advantages of using type names were discussed in Workbook 1, p. 3-11).

Fixed Point Type-Declarations -- Examples

Having made the following type-declarations:

```
TYPE CASH A 15, 7;

TYPE LG'NUM A 20, -10;
```

The following are correct uses of the type-names CASH and LG'NUM in an item-declaration:

```
ITEM SUM CASH;

ITEM MAXVAL LG'NUM = 2048;

CONSTANT ITEM TEST CASH = 16.374;

ITEM BASE LG'NUM;
```

FIXED POINT ITEM-DECLARATIONS -- EXERCISES

Given:

TYPE LARGE A 10,7;

Determine if the following declarations are correct or incorrect. If they are incorrect, make the appropriate corrections.

Declarations	C	I	Corrections
1. ITEM ROLL A 10, 3;			
2. ITEM SCRIPT A 6, -10;			
3. CONSTANT ITEM REAL VAL A			
4. CONSTANT ITEM XX A ,6 - 49;			
5. ITEM MASK LARGE;			
6. ITEM NUMBER LARGE A 20;			
7. ITEM TIME LARGE -6E4;			

ANSWERS

Declaration	C	I	Corrections
1. ITEM ROLL A 10, 3;	X		
2. ITEM SCRIPT A 6, -10;		X	ITEM SCRIPT A -6, 10; (scale plus fraction must be > 0
3. CONSTANT ITEM REAL VAL A 7;		X	CONSTANT ITEM REAL'VAL A 7 = 10.; (no blank in name, need preset)
4. CONSTANT ITEM XX A ,6 = 49;		X	CONSTANT ITEM XX A 8,6 = 49.; (need scale, real preset)
5. ITEM MASK LARGE;	X		
6. ITEM NUMBER LARGE A 20;		X	ITEM NUMBER LARGE; (need only one type)
7. ITEM TIME LARGE -6E4;		X	ITEM TIME LARGE = 6E4; (need =)

FIXED POINT TYPE EQUIVALENCE AND IMPLICIT CONVERSION

In order for an assignment-statement to be legal, the type of the source must be equivalent or implicitly convertible to the type of the target variable(s).

Two fixed point types are equivalent if their scale attributes are equal and their fraction attributes are equal.

A fixed point type will be implicitly converted to another fixed point type if the scale and fraction attributes of the target type are both at least as large as those of the source type.

Declarations

```
TYPE PAYMENT A 10, 7;
ITEM CASH PAYMENT;
ITEM MEASURE A 10, 7;
ITEM STATS A 10;
ITEM VISITS A 6, 2;
```

CASH and MEASURE have EQUIVALENT types - they are both A 10, 7.

If VISITS is the source in an assignment-statement and MEASURE is the target in the assignment-statement, the type of VISITS is IMPLICITLY CONVERTED to the type of MEASURE.

The type of STATS may or may not be either EQUIVALENT or IMPLICITLY CONVERTIBLE to the type of CASH.

ASSIGNMENT-STATEMENT: REVIEW EXERCISES

Determine if the following assignment-statements are correct or incorrect.

Declarations

```
TYPE SUM A 10, 6;

ITEM MAX SUM;

ITEM TOTAL A 5, 3;

ITEM MIN A 10, 6;

ITEM PLUS A 15;
```

Assignment-statements

1. MIN = TOTAL;
2. MIN, MAX = TOTAL;
3. TOTAL = PLUS;
4. PLUS = MAX;
5. PLUS, MAX, MIN = TOTAL;

ANSWERS

An assignment-statement assigns a source (formula) to a target (variable).

The type of the source must be equivalent or implicitly convertible to the type of the target.

An assignment-statement is either simple or multiple.

Declarations

```
TYPE SUM A 10, 6;
ITEM MAX SUM;
ITEM TOTAL A 5, 3;
ITEM MIN A 10, 6;
ITEM PLUS A 15;
```

Assignment-statements	C	I
1. MIN = TOTAL;	X	
2. MIN, MAX = TOTAL;	X	
3. TOTAL = PLUS;		X (scale of PLUS > scale of TOTAL)
4. PLUS = MAX;	X (if FIXED PRECISION >= 21)	
5. PLUS, MAX, MIN = TOTAL	X (if FIXED PRECISION >= 18)	

FIXED POINT EXPLICIT CONVERSION

An explicit conversion may be used to temporarily impose a template of one type on a value of another type. If the source in an assignment-statement is not equivalent or implicitly convertible to the type of the target(s) receiving the assignment, a conversion operator may be applied to the source to make the types compatible.

An explicit conversion consists of a conversion-operator applied to a parenthesized formula. An explicit conversion may have any of the following forms:

$$\left\{ \begin{array}{l} \text{(* type-description *)} \\ \text{type-indicator} \\ \text{item-type-name} \\ \text{(* item-type-name *)} \end{array} \right\} \text{(formula)}$$

Examples

```
(* A 10, 6 *)(COUNT)

CAST (COUNT)

        (where TYPE CAST A 4, -2;)

(* A 7 *) (COUNT)
```

> *NOTES: Type-indicator is a single letter only. The type-indicator A may not be used alone in an explicit conversion, as a scale must be specified for every fixed point item.*

ROUND-OR-TRUNCATE ATTRIBUTE

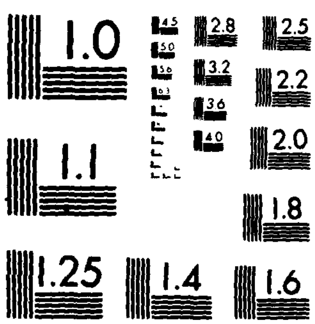
The round-or-truncate attribute specifies whether rounding or truncation is to occur when a value is converted to a fixed point type. If R is specified, rounding will occur. If T is specified, truncation towards minus infinity will occur. If the attribute is omitted, truncation in an implementation dependent manner will occur. (Rounding and truncation take place with respect to implemented precision.)

Examples

```
TYPE MINVAL A, T 5, 3;
TYPE MAXVAL A, R 10, 7;
```

AD-A108 527 SOFTECH INC WALTHAM MA F/G 5/9
THE JOVIAL (J73) WORKBOOK. VOLUME I. INTEGER AND FLOATING POINT--ETC(U)
NOV 81 F30602-79-C-0040
UNCLASSIFIED RADC-TR-81-333-VOL-1 NL

2 of 6
AD A108527


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0

EXPLICIT CONVERSION -- USES -- EXERCISES

Declarations

```
TYPE MAXVAL A,T 6,3;
TYPE MAXSUM U,T 10;
TYPE TOTAL F 30;
ITEM CANDY U 6;
ITEM COOKIES A 7,4;
ITEM CAKE F 20;
ITEM TART MAXVAL;
ITEM TASTE MAXSUM;
ITEM TANG TOTAL;
```

The following assignment-statements are incorrect. Apply the appropriate conversion operator to the source so the assignment may be made.

Incorrect	Correction
1. TART = COOKIES;	
2. CAKE = CANDY;	
3. TANG = COOKIES;	
4. TASTE = TANG;	
5. CANDY = TART;	
6. COOKIES = TASTE;	

ANSWERS

Declarations

```
TYPE MAXVAL A,T 6,3;
TYPE MAXSUM U,T 10;
TYPE TOTAL F 30;
ITEM CANDY U 6;
ITEM COOKIES A 7,4;
ITEM CAKE F 20;
ITEM TART MAXVAL;
ITEM TASTE MAXSUM;
ITEM TANG TOTAL;
```

	Incorrect	Correction
1.	TART = COOKIES;	TART = (*MAXVAL*)(COOKIES); TART = (*A 5,2*)(COOKIES);
2.	CAKE = CANDY;	CAKE = (*F 20*)(CANDY); if FLOATPRECISION <= 20 may use (*F*)
3.	TANG = COOKIES;	TANG = (*TOTAL*)(COOKIES); TANG = (*F, R 30*)(COOKIES);
4.	TASTE = TANG;	TASTE = (*MAXSUM*)(TANG); TASTE = (*U*)(TANG);
5.	CANDY = TART;	CANDY = (*U*)(TART); CANDY = (*U, T*)(TART);
6.	COOKIES = TASTE;	COOKIES = (*MAXVAL*)(TASTE); COOKIES = (*A 7,4*)(TASTE);

SECTION 2

DATA-DECLARATIONS:

BIT ITEMS

SOFTECH

DATA-DECLARATIONS: BIT ITEMS

Bit string values are sequences of binary values (bits). They may be used for communication with "ON/OFF" devices or to control parts of the program itself. For example, a bit string may be used to represent settings of switches on a control console.

Item-declarations for bit strings follow the general syntax for all items:

$$[\text{CONSTANT}] \ \text{ITEM} \begin{Bmatrix} \text{type-description} \\ \text{item-type-name} \end{Bmatrix} [\text{item-preset}];$$

ITEM	ITEM is a JOVIAL (J73) reserved word. It has a specific meaning to the compiler, which restricts its use to declarations. It cannot be used as a name.
name	A name is a sequence of letters, digits, dollar signs and primes (single quote characters). It must begin with a letter or a dollar sign, and be at least two characters long. A name may be any length, though only the first 31 characters are examined for uniqueness within the program. (Fewer than 31 characters may be examined for external names. External names are discussed in Workbook 8).[1]
type-description	The type-description for bit items consists of the letter B followed by an optional BIT-SIZE. (Bit-size is an integer compile-time formula that indicates how many bits are in the bitstring).
;	A semi-colon terminates the declaration.

BIT ITEM-PRESETS

A bit item may be optionally preset either by a bit literal or a Boolean literal.

Bit Literals: A bit literal is composed of a string of beads. The form of a bit literal is:

bead-size B 'bead...'

[1]See list of JOVIAL (J73) reserved words given in Workbook 1, p. 1:3-2..

Bead-size is the number of bits used to represent each bead. Bead-size may be 1 through 5, and must be large enough to represent each bead.

Examples	Meaning
1B'01101'	1 bit to represent 0, 1 bit to represent 1, ...; total of 5 bits
3B'07'	3 bits to represent 0, 3 bits to represent 7; total of 6 bits (000111 in binary)
4B'A0F'	4 bits to represent A, 4 bits to represent 0, 4 bits to represent F, (where A and F are hexadecimal numerals); total of 12 bits (101000001111 in binary)

Boolean Literals: A Boolean literal is a special form of a bit literal. A Boolean literal is always one bit long, and is either TRUE or FALSE. TRUE is represented as 1B'1'; FALSE is represented as 1B'0'. The form of the preset is:

= value

Bit Item-Presets: Examples

```
ITEM CODE B 6 = 1B'010100';

ITEM MASK B 16 = 4B'FFFF';

ITEM GAME B 12 = 3B'6411';

ITEM STAT B = FALSE;
```

CONSTANT BIT ITEMS

Bit items may be declared to have constant value. The form of a constant item declaration is:

$$\text{CONSTANT ITEM name} \begin{Bmatrix} \text{type-description} \\ \text{item-type-name} \end{Bmatrix} \text{item-preset;}$$

The JOVIAL (J73) reserved word CONSTANT and an item-preset are required.

Bit Item-Declarations: Examples

ITEM SWITCH B;	SWITCH is declared to be a bit item whose size is 1 bit by default.
ITEM FLAG B = TRUE;	FLAG is a bit item one bit long and preset to 1B'1'.
CONSTANT ITEM GUESS B7 = 1B'1100011';	GUESS is a constant bit item, seven bits in length, whose value will remain 1B'1100011' throughout the execution of the program.
ITEM ONES B9 = 3B'777';	ONES is a bit item preset to 9 ones.
ITEM MASK B8;	MASK is a bit item whose length is 8 bits.
ITEM CODE1 B;	CODE1 is declared to be a bit item whose length defaults to 1 bit.
ITEM SCAN B 16;	SCAN is a bit item whose length is 16 bits.

BIT TYPE-DECLARATIONS

A bit type-declaration allows the programmer to specify a template which can then be used to indicate type attributes in an item declaration. The syntax of the item type-description is as follows:

```
TYPE  item-type-name  type-description;
```

The item type declaration associates the item type-name with the attributes of the type-description.

> *NOTE: A type-declaration has no item-preset because it declares no data. A type-declaration creates a template. It defines what data described using a particular type-name will look like.*

Once a type-name has been declared, it may then be used in any item-declaration in place of an explicit type-description, as shown in the syntax below:

$$\text{[CONSTANT] ITEM name} \left\{ \begin{array}{l} \text{type-description} \\ \text{item-type-name} \end{array} \right\} \text{[item-preset];}$$

For example:

```
TYPE  ALPHA  B 16;
```

This type-declaration declares an item type-name ALPHA that may be used to declare bit items whose lengths are 16. ALPHA may be used as follows:

```
ITEM  OMEGA  ALPHA;
```

OMEGA is declared to be a bit item of length 16.

Bit Type-Declarations -- Examples

Having made the following type-declarations:

```
TYPE SUBMASK B 10;

TYPE HOLES B 6;
```

The following are correct uses of the item-type-names SUBMASK and HOLES in an item-declaration;

```
ITEM GAMES SUBMASK;

ITEM PRIME SUBMASK = 5B'98';

CONSTANT ITEM NOISE HOLES = 2B'301';

ITEM ESTIMATE HOLES;
```

BIT ITEM-DECLARATIONS -- EXERCISES

Write complete declarations for the following:

1. A type, CODE, to be 8 bits long and a type, MASK, to be 9 bits long.

2. An item, READING, of type CODE, to have all bits preset to zeros.

3. An item, WORD, of type MASK, to have each group of three bits preset to 1, 2, and 3.

4. An item, TIMES, of type MASK, to have all bits preset to ones.

5. A constant item, LAYERS, to be twelve bits long, and always to have its first 6 bits as 1's and its last 6 bits as 0's.

6. An item, FLAG, to be either TRUE or FALSE.

ANSWERS

```
1.   TYPE CODE B 8;
     TYPE MASK B 9;

2.   ITEM READING CODE = 1B'0';
     ITEM READING CODE = 4B'00';

3.   ITEM WORD MASK = 3B'123';

4.   ITEM TIMES MASK = 3B'111';

5.   CONSTANT ITEM LAYERS B 12 = 3B'7700';
     CONSTANT ITEM LAYERS B 12 = 2B'333000';

6.   ITEM FLAG B 1;
     ITEM FLAG B;
```

BIT TYPE EQUIVALENCE AND IMPLICIT CONVERSION

For an assignment statement to be legal, the type of its source must be equivalent or implicitly convertible to the type of the target variable(s).

Two bit types are equivalent if their size attributes are equal.

A bit string will be implicitly converted to any bit string with a different size attribute, with truncation on the left or padding with zeros on the left.

Declarations

```
TYPE MAXSTRING B 8;

ITEM SHORT B 4;

ITEM LONG MAXSTRING;

ITEM LARGE B 8;
```

LARGE and LONG have EQUIVALENT types - they are both B 8.

If SHORT is the source in an assignment-statement and LONG is the target in the assignment-statement, SHORT will be right-justified in its field and padded on the left with zeros, (i.e., the type of SHORT is IMPLICITLY CONVERTED to the type LONG).

If LONG is the source in an assignment-statement and SHORT is the target in the assignment-statement, LONG will be truncated on the left to be the same length as SHORT, (i.e., the type of LONG is IMPLICITLY CONVERTED to the type of SHORT).

EXERCISES

Determine if the following declarations and statements are correct. Make the appropriate corrections wherever necessary.

Exercises	C	I	Corrections
1. TYPE LONG$ B 20			
2. TYPE NO BITS B 3;			
3. ITEM ALPHA B 7 = 1B'10';			
4. ITEM BETA B 2 = TRUE;			
5. CONSTANT ITEM GAMMA NOBITS;			
6. ALPHA = BETA;			
7. GAMMA = BETA;			
8. BETA = ALPHA OR GAMMA;			
9. ALPHA,BETA = FALSE;			
10. BETA = 6B'1';			

ANSWERS

Exercises	C	I	Corrections
1. TYPE LONG$ B 20		X	TYPE LONG$ B 20;
2. TYPE NO BITS B 3;		X	TYPE NOBITS B 3;
3. ITEM ALPHA B 7 = 1B'10';	X		
4. ITEM BETA B 2 = TRUE;	X		
5. CONSTANT ITEM GAMMA NOBITS;		X	CONSTANT ITEM GAMMA NOBITS = 3B'0';
6. ALPHA = BETA;	X		
7. GAMMA = BETA;		X	(may not assign to CONSTANT)
8. BETA = ALPHA OR GAMMA;	X		
9. ALPHA, BETA = FALSE;	X		
10. BETA = 6B'1';		X	(bead-size too large)

BIT TYPE EXPLICIT CONVERSION

A value of any type may be explicitly converted to a bit type.

The forms of an explicit conversion are:

$$\left\{\begin{array}{l}\text{(* type-description *)}\\ \text{type-indicator}\\ \text{item-type-name}\\ \text{(* item-type-name *)}\end{array}\right\}\text{(formula)}$$

When a value of any type is explicitly converted to type B n, the result is the rightmost n bits of the value. If the target is longer than the source, the value will be right-justified and padded on the left with zeros. If the source is longer than the target, truncation will occur on the left.

Examples

(* B 12 *) (FLAGS)

B (FLAGS)

SHORTSTRING (FLAGS)

(where TYPE SHORTSTRING B 7;)

SECTION 3

DATA-DECLARATIONS: CHARACTER ITEMS

SOFTECH

DATA-DECLARATIONS: CHARACTER ITEMS

A character item is a fixed length string of characters. A character item is declared using the customary syntax for item-declarations:

$$\text{[CONSTANT] ITEM name} \begin{Bmatrix} \text{type-description} \\ \text{item-type-name} \end{Bmatrix} \text{[item-preset];}$$

ITEM	ITEM is a JOVIAL (J73) reserved word. It has a specific meaning to the compiler, which restricts its use to declarations. It cannot be used as a name.
name	A name is a sequence of letters, digits, dollar signs and primes (single quote characters). It must begin with a letter or a dollar sign, and be at least two characters long. A name may be any length, though only the first 31 characters are examined for uniqueness within the program. (Fewer than 31 characters may be examined for external names. External names are discussed in Workbook 8) [1]
type-description	A character type has a type-description of C [character-size] where character-size is an integer formula known at compile time and less than or equal to the implementation parameter MAXBYTES. If character-size is not specified, the default size is 1.
;	A semi-colon terminates the declaration.

CHARACTER ITEM-PRESETS

A character item declaration may contain an optional item preset. The form of the preset is:

= value

where value is a character literal.

---

[1]See list of JOVIAL (J73) reserved words given in Workbook 1, p. 1:3-2.

A character literal is a sequence of characters enclosed in primes. These characters can be letters, numbers or special characters. The form is:

'character ...'

Every character, including blanks, counts as one character. To get a prime (') to appear in a character string, it is doubled. However, it only counts as one character (i.e., 'I''m here').

CONSTANT CHARACTER ITEMS

A character item may be declared to be constant, in which case its value cannot be altered during execution. The form is:

$$\text{CONSTANT ITEM name} \left\{ \begin{array}{l} \text{type-description} \\ \text{item-type-name} \end{array} \right\} \text{item-preset;}$$

The reserved word CONSTANT and an item-preset are required.

Character Item-Declarations: Examples

ITEM NAME C 20;	NAME is declared to be a character item of 20 characters in length.
ITEM NAME2 C 7 = 'GANDALF';	NAME2 is a character item of length 7 with a preset value of GANDALF.
CONSTANT ITEM STRING C13 = '014ABX+*/''123'	STRING is a constant character item of length 13, whose value will always be 014ABX+*/'123 during program execution.

CHARACTER TYPE-DECLARATIONS

A character type-declaration allows the programmer to specify a template which can then be used to indicate type attributes in an item declaration. The syntax of the item type-description is as follows:

```
TYPE  item-type-name  type-description;
```

The item type-declaration associates the item type-name with the attributes of the type-description.

> *NOTE: A type-declaration has no item-preset because it declares no data. A type-declaration creates a template. It defines what data described using a particular type-name will look like.*

Once a type-name has been declared, it may then be used in <u>any</u> item declaration in place of an explicit type-description, as shown in the syntax below:

$$\text{[CONSTANT] ITEM name} \left\{ \begin{array}{l} \text{type-description} \\ \text{item-type-name} \end{array} \right\} \text{[item-preset];}$$

For example:

```
TYPE  ALPHA  C 10;
```

This type-declaration declares an item type-name ALPHA that may be used to declare character strings whose lengths are 10. ALPHA may be used as follows:

```
ITEM  OMEGA  ALPHA;
```

OMEGA is declared to be a character item of length 10.

Having made the following type-declarations:

```
TYPE LONGCHAR C 20;

TYPE MEDCHAR C 15;
```

The following are correct uses of the item-type-names LONGCHAR and MEDCHAR in an item declaration:

```
CONSTANT ITEM SOC'SEC'NUM MEDCHAR = '391-56-0291';

ITEM LAST'NAME LONGCHAR = 'BEETHOVEN';

ITEM ADDRESS LONGCHAR;
```

CHARACTER ITEM-DECLARATIONS -- EXERCISES

What is wrong with the following declarations:

1. CONSTANT ITEM ADDR HOME C 20 - 'WALTHAM';
2. ITEM MY.JOB C 10;
3. ITEM WEATHER = 'HOT AND SUNNY';
4. TYPE SNOW C 3 = 'WET';
5. ITEM JOB C 12 = TEACHER;
6. CONSTANT ITEM BOOKS C 20 = 'I'M OK, YOU'RE OK';

ANSWERS

What is wrong with the following declarations:

1. CONSTANT ITEM ADDR HOME C 20 = 'WALTHAM';

   (blank not allowed in name)

2. ITEM MY.JOB C 10;

   (. not allowed in name)

3. ITEM WEATHER = 'HOT AND SUNNY';

   (need type - ITEM WEATHER C 20 = "HOT AND SUNNY';)

4. TYPE SNOW C 3 = 'WET';

   (no preset allowed on TYPE)

5. ITEM JOB C 12 = TEACHER;

   (preset is character literal, must be in single quotes)

6. CONSTANT ITEM BOOKS C 20 = 'I'M OK, YOU'RE OK';

   (need 2 single quotes inside character literal)

CHARACTER TYPE EQUIVALENCE AND IMPLICIT CONVERSION

For an assignment-statement to be legal, the type of the source must be equivalent or implicitly convertible to the type of the target variable(s).

Two character types are equivalent if their size attributes are equal.

A character string will be implicitly converted to a character string with a different size attribute with truncation on the right or padding with blanks on the right.

Declarations

```
TYPE LAST'NAME C 20;

ITEM COMPOSER C 20;

ITEM AUTHOR LAST'NAME;

ITEM BOOK C 10;
```

AUTHOR and COMPOSER have *EQUIVALENT types - they are both* C 20's.

If AUTHOR is the source of an assignment-statement and BOOK is the target in the assignment-statement, AUTHOR will be truncated on the right to be the same length as BOOK, (i.e., the type of AUTHOR is IMPLICITLY CONVERTED to the type of BOOK).

If BOOK is the source of an assignment-statement and COMPOSER is the target in the assignment-statement, BOOK will be padded on the right with blanks, (i.e., the type of BOOK will be IMPLICITLY CONVERTED to the type of COMPOSER).

CHARACTER TYPE EXPLICIT CONVERSION

The forms of an explicit conversion are:

$$\left\{\begin{array}{l}\text{(* type-description *)}\\ \text{type-indicator}\\ \text{item-type-name}\\ \text{(* item-type-name *)}\end{array}\right\}\text{(formula)}$$

When a character formula is explicitly converted to type C n, the result is the leftmost n characters of the character formula. If the target is longer than the source, the formula is padded on the right with blanks. If the source is longer than the target, the formula is truncated on the right.

Examples

(* C 20 *) (NAME)

C (NAME)

LONGNAME (NAME)

(where TYPE LONGNAME C 30;)

Conversion of a bit string to a character type is legal only if the size of the bit string is equal to the actual number of bits used to represent the character type (excluding filler bits between bytes) which may be found by using the BITSIZE function. The number of bits in the character string, including filler bits, may be found using the REP conversion.

SECTION 4

DATA-DECLARATIONS:
STATUS ITEMS

DATA-DECLARATIONS: STATUS ITEMS

A status item is an item whose value range is a specified list of symbolic names called status-constants. A status-constant is a symbolic constant that has an ordering relation with the other status-constants in the list. A status item provides a way to enumerate all possible values for that item in a mnemonic way. For example, the IRS asks each person to declare a filing status - single, married-filing jointly, married-filing separately, etc. A programmer could declare a variable:

```
ITEM FILING U;
```

and somewhere make the notation that FILING = 1 means single, FILING = 2 means married-filing jointly, FILING = 3 means married-filing separately, etc.

Another way would be to declare a STATUS item, and list all possible values for that item in a mnemonic way:

```
ITEM FILING STATUS ( V( SINGL ), V( MAR'JT ),
      V( MAR'SEP ), ... );
```

This STATUS item-declaration explicitly enumerates all the possible values for the item FILING.

> *NOTE: The use of a name in a status constant does not constitute a declaration of the name or a reference to a declared name with the same spelling. For example, the status-constant V(MONDAY) declares the name V(MONDAY) not MONDAY. A status name and a declared name with the same spelling can exist in the same scope without any conflict.*

A status item is declared using the now familiar general syntax for item declarations:

```
[CONSTANT] ITEM name  type-description  [item-preset];
                      item-type-name
```

ITEM	ITEM is a JOVIAL (J73) reserved word. It has a specific meaning to the compiler, which restricts its use to declarations. It cannot be used as a name.
name	A name is a sequence of letters, digits, dollar signs and primes (single quote characters). It must begin with a letter or a dollar sign, and be at least two characters long. A name may be any length, though only the first 31 characters are examined for uniqueness within the program. (Fewer than 31 characters may be examined for external names. External names are discussed in Workbook 8). [1]
type-description	A default status type has a type description of the following form STATUS (status-constant, ...) and a status-constant has the form $V(\begin{Bmatrix} \text{name} \\ \text{letter} \\ \text{reserved-word} \end{Bmatrix})$ The name must be unique for each constant. Status-constants are represented by the compiler as the values 0 → N-1, where N is the number of status-constants in the list. (These are the default representations of the status-constants.) However, the programmer treats the status item as an item which may only take on the enumerated values. The item is not treated as an integer. Even though the representation of a status-constant is an integer, an integer value may not be assigned to a status item. The size of a status item is the minimum number of bits necessary to hold the representation of the status-constant with the largest representation.
;	A semi-colon terminates the declaration.

STATUS ITEM-PRESETS

A status item declaration may optionally include a preset. The form of the preset is

= **value**

---

[1]See list of JOVIAL (J73) reserved words given in Workbook 1, p. 1:3-2.

where value is a status-constant from the list enumerated in the item-declaration.

CONSTANT STATUS ITEMS

A status item may be declared to be constant, in which case its value cannot be altered during execution. The form is

$$\text{CONSTANT ITEM name} \left\{ \begin{array}{l} \text{type-description} \\ \text{item-type-name} \end{array} \right\} \textbf{item-preset;}$$

The reserved word CONSTANT and an item preset are required.

Status Item Declarations: Examples

ITEM SEASON STATUS (V(SPRING), V(SUMMER), V(FALL), V(WINTER))	Item SEASON is declared to be a status item which can assume the values V(SPRING), V(SUMMER), V(FALL), V(WINTER). The size of SEASON is 2 bits, since the largest representation is binary 3 (11) which is required for V(WINTER).
ITEM GRADES STATUS (V(A), V(B), V(C), V(D), V(F));	GRADES is a status item which can assume the values V(A), V(B), V(C), V(D), and V(F). The size of grades is 3 bits, since binary 4 is the largest representation in the list.
ITEM WORDS STATUS (V(ITEM), V(STATUS), V(BEGIN)) = V(BEGIN);	WORDS is a status item which can assume the values V(ITEM), V(STATUS), and V(BEGIN). WORDS is preset to V(BEGIN).
CONSTANT ITEM LIGHT STATUS (V(RED), V(YELLOW), V(GREEN)) = V(RED);	LIGHT is a constant status item whose value is V(RED).

STATUS TYPE-DECLARATIONS

A status type-declaration allows the programmer to specify a template which can then be used to indicate type attributes in an item-declaration. The syntax of the item type description is as follows:

TYPE item-type-name type-description;

The item type-declaration associates the item type name with the attributes of the type-description.

> *NOTE: A type-declaration has no item-preset because it declares no data. A type-declaration creates a template. It defines what data described using a particular type-name will look like.*

Once a type-name has been declared, it may then be used in <u>any</u> item declaration in place of an explicit type-description, as shown in the syntax below:

[CONSTANT] ITEM name $\left\{ \begin{array}{l} \text{type-description} \\ \textbf{item-type-name} \end{array} \right\}$ [item-preset];

For example:

TYPE ALPHA STATUS (V(READY), V(SET), V(GO), V(STOP));

This type-declaration declares an item type-name ALPHA that may be used to declare a status item whose enumerated values are V(READY), V(SET), V(GO), V(STOP). ALPHA may be used as follows:

ITEM OMEGA ALPHA;

Having made the following type-declarations:

TYPE LETTERS STATUS (V(X), V(Y), V(Z));

TYPE COLORS STATUS (V(BLACK), V(WHITE));

The following are correct uses of the item-type-names LETTERS and COLORS in an item-declaration:

```
ITEM ENDALPHA LETTERS = V(Z);

ITEM SHADES COLORS;

CONSTANT ITEM VARX LETTERS = V(X);

ITEM MAGIC COLORS;
```

SPECIFIED STATUS LISTS

A specified status list is one in which the representational values (status-index) associated with the status-constants (by the compiler) can be specified by the programmer. The number of bits used to hold these representations (size) may also be specified. Specified status lists are commonly used to match up with hardware devices.

The general form of a status type-description is:

STATUS [size] (status-group, ...)

Each status-group has the form:

[status-index] status-constant

A status-constant is of the form:

$$V\ \left(\begin{Bmatrix}\text{name}\\ \text{letter}\\ \text{reserved-word}\end{Bmatrix}\right)$$

Specified Status List -- Example

Given the following declaration:

```
TYPE GRADE STATUS 5(1V(A), 2V(B), 4V(C), V(D), 9V(F));
```

The size of 5 indicates that the largest representation for a status-constant in this list may be only 5 bits long.

The status-index of 4 on V(C) indicates that V(C) is represented as 4.

NOTES: If a size is not given, the minimum number of bits necessary to represent the largest representation is used.

STATUS LISTS -- EXERCISES

Given the following declarations:

TYPE COLOR STATUS (V(RED), V(ORANGE), V(YELLOW));

TYPE NAME STATUS (V(SUE), 3V(MARY), V(FRANK), 9V(TOM));

TYPE DOGS STATUS 6(2V(RETRIEVER), 7V(SPANIEL), 3V(COLLIE));

How many bits will be needed to represent the status-constants in:

COLOR

NAME

DOGS

What is the representation of:

V(RED)

V(FRANK)

V(SPANIEL)

V(SUE)

ANSWERS

Given the following declarations:

TYPE COLOR STATUS (V(RED), V(ORANGE), V(YELLOW);

TYPE NAME STATUS (V(SUE), 3V(MARY), V(FRANK), 9V(TOM));

TYPE DOGS STATUS 6(2V(RETRIEVER), 7V(SPANIEL), 3V(COLLIE));

How many bits will be needed to represent the status-constants in:

COLOR	2
NAME	4
DOGS	6

What is the representation of:

V(RED)	0
V(FRANK)	4
V(SPANIEL)	7
V(SUE)	0

STATUS TYPE EQUIVALENCE AND IMPLICIT CONVERSION

For an assignment-statement to be legal, the type of the source must be equivalent or implicitly convertible to the type of the target variable(s).

Two status types are equivalent if they both have default representation, if their size attributes are the same, and if both status-lists contain the same status-constants in the same order; or, if both have identical specified representations, their size attributes are the same, and both status-lists contain the same status-constants.

A status type will be implicitly converted to a status type that differs only in its size attribute.

Examples

Declarations

```
TYPE GRADES STATUS (V(A), V(B), V(C));

ITEM LETTERS STATUS (V(A), V(B), V(C));

ITEM ABC GRADES;

ITEM ALPHA STATUS 5(V(A), V(B), V(C));

ITEM SPEC'ALPHA STATUS 5(3V(A), 7V(B), 20V(C));
```

ABC and LETTERS have EQUIVALENT types - their status-lists are identical.

The type of LETTERS may be IMPLICITLY CONVERTED to the type of ALPHA - the status-lists differ only in their size attributes.

The type of SPEC'ALPHA will not be IMPLICITLY CONVERTED to the type of ABC - the representations are different.

DISAMBIGUATING STATUS-CONSTANTS

Given the following declarations:

```
TYPE BED STATUS(V(SPRING), V(WATER), V(AIR));

TYPE SEASON STATUS (V(WINTER), V(SPRING), V(SUMMER),
V(FALL));
```

ITEM SLEEP BED;

ITEM CLIMATE SEASON;

The status-constant V(SPRING) is a part of two type-declarations. Used by itself, it is said to be ambiguous, and an explicit conversion will have to be applied. V(SPRING) may be automatically disambiguated (implicitly converted) to one type or another depending on the context.

A status-constant that belongs to more than one status-list is automatically disambiguated in the following contexts, some of which will be discussed in later workbooks:

- When it is the source value of an assignment-statement, it takes the type of the target variable.
- When it is an actual parameter, it takes the type of the corresponding formal parameter.
- When it is in a table subscript or used in a preset to specify an index, it takes the type of the corresponding dimension in that table's declaration.
- When it is a loop initial-value, it takes the type of the loop-control variable.
- When it is an item-preset or table-preset, it takes the type of the item or table item being initialized.
- When it is an operand of a relational operator, it takes the type of the other operand.
- When it is in a case-index-group, it takes the type of the case-selector.
- When it is a lower-bound or upper-bound, it takes the type of the other bound.

Examples	Type of V(SPRING)
SLEEP = V(SPRING);	BED
IF CLIMATE < V(SPRING);	SEASON
(V(SPRING) : V(AIR))	BED(table dimension)

STATUS TYPE EXPLICIT CONVERSION

The forms of an explicit conversion are:

$$\left\{\begin{array}{l}\text{(* type-description *)}\\ \text{type-indicator}\\ \text{item-type-name}\\ \text{(* item-type-name *)}\end{array}\right\}\ \text{(formula)}$$

A status-conversion is used to explicitly convert a data object to a status type. The conversion can be applied to bit or status data objects only.

A bit string will be treated as representing the representational value of the status type if the size of the bit string equals the BITSIZE of the status type and the value of the bit string is within the range of values of the status type. Otherwise the conversion is illegal.

A status-conversion may be used to assert the type of a status object. This will be required when a status constant belongs to more than one type and it is used in a context other than these enumerated above under implicit conversions. Except for status objects whose types differ only in their size attributes, a status object cannot be converted to a different status type without first converting it to a bit string.

In the example on the previous page, V(SPRING) could be explicitly disambiguated in either of the following manners:

```
(* BED *) (V(SPRING))

BED(V(SPRING))
```

SECTION 5

FORMULAE

SOFTECH

FORMULAE

A formula is either a single operand or a combination of operators and operands. A formula may be parenthesized and the result of a formula has a type. This section discusses bit and fixed point formulae. There are no character or status formulae defined in JOVIAL (J73).

OPERATOR PRECEDENCE: REVIEW

The order in which operators and operands are combined is determined by the precedence of the operators. Operators of equal precedence are evaluated from left to right, if the operation is not commutative (i.e., in addition, the order of evaluation is determined by the most efficient code which can be generated, since 5 + 3 is the same as 3 + 5). **Expressions within parenthesis are evaluated first, from the innter most parenthesis out.**

A complete list of JOVIAL (J73) operators and their precedence is as follows:

Precedence	Operator
6	@, subscripting, function calls
5	**
4	*, /, MOD
3	+, -
2	=, <>, <, >, <=, >=
1	NOT, AND, OR, EQV, XOR
0	assignment

FIXED POINT FORMULAE

A fixed point formula must have fixed point operands or one fixed operand and one integer. A fixed point formula may use the following operands:

+ - * /

Modulus and exponentiation are undefined. For example, given these declarations:

```
ITEM FIRST' A 10,6;

ITEM SECOND' A 10,6;
```

The following are valid formulae:

```
FIRST' + SECOND'

FIRST' - SECOND'

FIRST' * SECOND'

FIRST' / SECOND'

FIRST' - ( SECOND' - FIRST' )
```

THE TYPE OF A FIXED POINT FORMULA

The type of the result of a fixed point formula depends on the operator.

Operator	Result Type
+ -	Operands must have the same scale. Scale of the result is the scale of the operands. Fraction of the result is fraction of the operand with the larger fraction.

Declarations

```
ITEM FIRST A 10, 6;

ITEM SECOND A 10,2;

ITEM THIRD A 6,3;
```

Formulae	Result Type
FIRST - SECOND	A 10,6
SECOND + SECOND	A 10,2
SECOND + THIRD	cannot do as written, could do (* A 6, 3 *) (SECOND) + THIRD or (SECOND) + (* A 10,2 *) (THIRD)
FIRST-(SECOND - FIRST)	A 10,6

Operator	Result Type
*	If one operand is an integer, the scale and the fraction of the result are the same as the scale and the fraction of the fixed point operand. If both operands are fixed, the scale and the fraction of the result are the sum of the scale and the fraction, respectively, of the operands.

Declarations

```
ITEM FIRST A 10,6;

ITEM SECOND A 8,-3;

ITEM THIRD S 10;
```

Formulae	Result Type
FIRST * SECOND	A 18,3
SECOND * THIRD	A 8,-3
FIRST * FIRST	A 20,12

> *NOTE: If the precision (sum of the scale and fraction) of the result exceeds MAXFIXEDPRECISION, or if the scale does not lie within -127→+127, an explicit conversion must be applied to the result to yield a valid scale and precision.*

Operator	Result Type
/	If the divisor is an integer, the scale and the fraction of the result are the same as the scale and the fraction of the fixed point operand. Otherwise, the result must be explicitly converted to a specific scale and fraction.

Declarations

```
TYPE ANSWER A 8,7;

ITEM FIRST A 10,6;

ITEM SECOND A 8,-3;

ITEM THIRD S 10;
```

Formulae	Result Type
FIRST / THIRD	A 10,6
SECOND / THIRD	A 8,-3
THIRD / SECOND	must be explicitly converted (* A 12, 4 *) (THIRD / SECOND)
SECOND / FIRST	must be explicitly converted ANSWER (SECOND / FIRST)

EXERCISES

> Remember: expressions within parenthesis are evaluated first, ** (exponentiation) second, * / MOD third, + - fourth.
>
> Operators of the same precedence level are evaluated left to right.

Determine if the following formulae are correct. If they are, give the resulting value and type.

Declarations

```
TYPE CHARGE A 6,4;
TYPE MASTER U 7;
ITEM BANANCE A 6,03 = 24.;
ITEM PROFITS A -2,7 = .125;
ITEM TIME CHARGE = -1.1;
ITEM FACTOR MASTER = 4;
```

Formulae	C	I	Value	Type
1. TIME + BALANCE				
2. PROFITS + BALANCE + 17				
3. PROFITS / BALANCE + 49.6				
4. (FACTOR + 1) * PROFITS				
5. FACTOR ** TIME				

ANSWERS

Declarations

```
TYPE CHARGE A 6,4;
TYPE MASTER U 7;
ITEM BALANCE A 6,03 = 24.;
ITEM PROFITS A -2,7 = .125;
ITEM TIME CHARGE = -1.1;
ITEM FACTOR MASTER = 4;
```

Formulae	C	I	Value	Type
1. TIME + BALANCE	X		22.9	A 6,4
2. PROFITS + BALANCE + 17		X (need real literal, same scales)		
3. PROFITS / BALANCE + 49.6		X (need explicit conversion in division)		
4. (FACTOR + 1) * PROFITS	X		.625	A -2,7
5. FACTOR ** TIME		X (** not allowed with fixed)		

BIT FORMULAE

A bit formula is a formula whose operands are bit strings. If the number of bits in the two operands is not equal, the smaller operand is padded on the left with zero bits until they are the same size. The operators (see Figure 5-1) used in a bit formula are:

NOT AND OR XOR EQV

Parentheses are necessary to indicate the order of evaluation if the formula has more than one kind of logical operator. Otherwise, the formula will be evaluated left to right.

Bit Formulae -- Examples

Declarations

```
ITEM MASK B 6 = 1B'001100';
ITEM CODE B 6 = 1B'011011';
```

Formulae	Results
NOT MASK	1B'110011'
MASK OR CODE	1B'011111'
MASK XOR CODE	1B'010111'
MASK AND CODE	1B'001000'
MASK EQV CODE	1B'101000'
MASK AND (CODE XOR MASK)	1B'000100'

AND: If both operands '1', Result = 1;

If either operand zero, Result = 0

result	left	right
0	0	0
0	0	1
0	1	0
1	1	1

result = left AND right

result	left	right
0	0	0
1	0	1
1	1	0
1	1	1

result = left OR right

OR: If either operand is 1, Result = 1

XOR: If both operands the same, Result = 0; Else, result is 1.

result	left	right
0	0	0
1	0	1
1	1	0
0	1	1

result = left XOR right

result	left	right
1	0	0
0	0	1
0	1	0
1	1	1

result = left EQV right

EQV: If both operands identical, Result = 1

result	left
1	0
0	1

result = NOT left

NOT: Result opposite of operand.

Figure 5-1. Logical Operators

SECTION 6

SUMMARY

ITEM-DECLARATIONS

All data items must be declared before they may be used anywhere in a program. The general form of an item-declaration is:

$$\text{[CONSTANT] ITEM name} \begin{Bmatrix} \text{type-description} \\ \text{item-type-name} \end{Bmatrix} \text{[item-preset] ;}$$

The type-descriptions shown so far are:

U [integer-size]

S [integer-size]

F [precision]

A scale [,fraction]

B [bit-size]

C [character-size]

$$\text{STATUS [size] ([status-index] V (} \begin{Bmatrix} \text{name} \\ \text{letter} \\ \text{reserved-word} \end{Bmatrix} \text{), ...)}$$

Any literal value (any formula known at compile-time) may be used as an item-preset. The form is:

ITEM name type-description item-preset ;

The type of an item-preset must be equivalent or implicitly convertible to the type of the item. Otherwise, an explicit conversion operator must be applied. The form of an item-preset is:

= value

An item may be declared to be constant. The form is:

CONSTANT ITEM name type-description item-preset ;

A programmer may make a type-declaration. The form is:

TYPE item-type-name type-description ;

The item-type-name may then be used in an item-declaration:

ITEM name item-type-name ;

A more general form of an item-declaration is:

[CONSTANT] ITEM name $\left\{\begin{array}{l}\text{type-description}\\ \text{item-type-name}\end{array}\right\}$ [item-preset] ;

FORMULAE

A formula is either a single operand or a combination of operators and operands, possible parenthesized. An operator may be either prefix (+, -, NOT) or infix (all operators).

Type of Operands	Operators	Result Type
U, S	+ - * / MOD	S n, where n is one less than the multiple of BITSINWORD to hold the larger operand; operands may be of either type, any size.
	**	Only if the right operand is non-negative, compile-time known, result is S n (as above); all other cases handled as floating point.
F	+ - * / **	F p, where p is the precision of the larger operand; operands may be of any precision.
A	+ - * /	Type of result and type of operands depend on choice of operator:
		+, - Operands must have same scale; result if A s [,f] where f is larger fraction.
		* If one operand is fixed and one is integer, result is type of fixed operand: A s [,f].
		* If both operands are fixed, result type is A $s_1 + s_2$ [,$f_1 + f_2$].
		/ If divident is fixed and divisor is integer, result type is type of fixed operand.

Type of Operands	Operators	Result Type
		/ Otherwise, result type must be explicitly converted.
B	NOT AND OR XOR EQV	B n, where n is the size of the longer bit string; operands may be of any size.
C	none	None.
STATUS	none	None.

EXPLICIT CONVERSION

The form of an explicit conversion operator is:

$$\left\{\begin{array}{l}\text{(* type-description *)}\\ \text{type-indicator}\\ \text{item-type-name}\\ \text{(* item-type-name *)}\end{array}\right\} \quad \text{(formula)}$$

> *NOTE: The type-indicator A (fixed point) may not be used alone as a conversion operator.*

TYPE MATCHING, CONVERSION -- SUMMARY

Target Type =	Source Type: Equivalent	Source Type: Implicitly Convertible	Source Type: Explicitly Convertible
U	U same size	U, S any size	U, S any size
			F any precision
			A any scale [,fraction]
			B smaller or same BITSIZE
S	S same size	U, S any size	U, S any size
			F any precision
			A any scale [,fraction]
			B smaller or same BITSIZE
F	F same precision	F smaller precision	U, S any size
			F any precision
			A any scale [,fraction]
			B same BITSIZE
B	B same size	B any size	U, S any size
			F any precision
			A any scale [,fraction]
			B any size
			C any size
			STATUS any size

Target Type =	Source Type: Equivalent	Implicitly Convertible	Explicitly Convertible
A	A same scale [,fraction]	A smaller scale [,smaller fraction]	U, S any size F any precision A any scale [,any fraction] B same BITSIZE
C	C same size	C any size	B same BITSIZE C any size
STATUS	STATUS same representation, same size-attributes, same status-constants	STATUS same representation same status-constants different size attributes	B same BITSIZE* STATUS to disambiguate

TYPE MATCHING, CONVERSION -- SUMMARY NOTES

The programmer has added control when using the round-or-truncate attribute for numeric conversions.

Bit strings are padded on the left with zeros or truncated on the left to obtain the appropriate length.

Character strings are padded on the right with blanks or truncated on the right to obtain the appropriate length.

When assigning a large size integer into a smaller size integer, the results are unpredictable.

The type-indicator A may not be used alone as a conversion operator, since all fixed point declarations must specifiy a scale.

---

*If the value is represented in the status-list.

If the programmer feels it is necessary to convert, for example, an integer to a character string, this may be done with a two-step conversion:

<u>Example</u>

```
ITEM NUMBER U 32;
ITEM NAME C 4;
NAME + (* C 4 *)  (( * B 32 * )  (NUMBER));
```

ASSIGNMENT-STATEMENTS

An assignment-statement assigns a source (value) to one or more targets (variables). The type of the source must be equivalent or implicitly convertible to the type of the target.

The form an an assignment-statement is:

```
variable, ... = value;
```

ITEM-DECLARATIONS -- EXERCISES

Declare items for the following:

1. A type, COUNT, that can take on an integer value in the range 0 through 30.

2. A type, NAME, that consists of eleven characters.

3. A number, PRICE, that has six binary places to the left of the binary point and three binary places to the right.

4. A condition, ALERT, that can be RED, YELLOW, or GREEN.

5. A switch, SWITCH, that consists of 4 bits.

6. A counter, EPSILON, that can have integer values in the range -1000 through 1000 and that has the initial value 0.

7. A name, AUTHOR, to be of the type declared in #2.

8. An index, INDEX, to be of the type declared in #1.

ANSWERS

1. TYPE COUNT U 5;
   TYPE COUNT U;
   TYPE COUNT S;

2. TYPE NAME C 11;

3. ITEM PRICE A 6,3;

4. ITEM ALERT STATUS (V(RED), V(YELLOW), V(GREEN));

5. ITEM SWITCH B 4;

6. ITEM EPSILON S 10 = 0;
   ITEM EPSILON S = 0;

7. ITEM AUTHOR NAME;

8. ITEM INDEX COUNT;

MACHINE LIMITATIONS

Every implementation has limits on the sizes and precisions allowable for declared items. These limits are defined in terms of machine parameters. The restrictions are:

for U [integer-size]

1	< =	integer-size	< =	MAXINTSIZE

for S [integer-size]

1	< =	integer-size	< =	MAXINTSIZE

for F [precision]

1	< =	precision	< =	MAXFLOATPRECISION

for A scale [,fraction]

-127	< =	scale	< =	127

for B [bit-size]

1	< =	bit-size	< =	MAXBITS

for C [character-size]

1	< =	character-size	< =	MAXBYTES

VALUE LIMITATIONS

Given the size or precision of an item, a programmer may determine the range of values that item may have. This is done by using machine parameter functions. The value returned by any of these machine parameter functions is a constant and may be used anywhere a constant may be used. The functions are:

for U [integer-size]

	0 < =	value	< =	MAXINT (integer-size)

for S [integer-size]

MININT (integer-size)	< =	value	< =	MAXINT (integer-size)

for F [precision]

MINFLOAT (precision)	< =	value	< =	FLOATUNDERFLOW (precision)
or	0 < =	value		
or FLOATUNDERFLOW (precision)	<=	value	< =	MAXFLOAT (precision)

for A scale [fraction]

MINFIXED (scale, fraction)	< =	value	< =	MAXFIXED (scale, fraction)

THE JOVIAL (J73) WORKBOOK

VOLUME 3

EXECUTABLE STATEMENTS

1081-1

April 1981

Submitted to

Department of the Air Force
Rome Air Development Center
I S I S
Griffiss Air Force Base, NY 13441

Prepared by

SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02154

PREFACE

Workbook 3 is intended for use with Tape 3 of the JOVIAL (J73) Video Course. Its purpose is to elaborate upon and reinforce concepts and language rules introduced in the videotape.

Simple and compound executable statements are discussed in this workbook. Specifically addressed are assignment-statements, relational expressions, if-statements, goto's, case-statements, while-loops, for-loops and exit-statements. The final section is a summary of the information presented in this segment.

TABLE OF CONTENTS

SYNTAX

The syntax conventions used in the JOVIAL (J73) Video Course and workbook are as follows:

Syntax	Meaning	Sample Expansions
[some-feature]	Brackets indicate an optional feature.	some-feature OR nothing
{one | other}	Braces with a vertical bar indicate disjunction- a choice between alternatives.	one OR other
{this-one that-one}	Braces with each feature on separate lines indicate disjunction - a choice between alternatives.	this-one OR that-one
letter ...	The sequence '...' indicates one or more repetitions of a feature.	letter letter letter letter letter letter ...
(letter),...	The sequence "..." following a comma (or a colon) indicates one or more repetitions of the feature separated by commas (or colons).	(letter) (letter) (letter) (letter) (letter) (letter) ...
[{this-one that-one}] + another	Syntax symbols may be combined.	this-one + (another) that-one + (another) + (another)

SECTION 1

ASSIGNMENT-STATEMENTS

SOFTECH

ASSIGNMENT-STATEMENT

Statements are the means by which algorithms are specified. Thus far, the declaration-statement and the assignment-statement have been discussed. While the declaration-statement is a non-executable statement, the assignment-statement specifies an action to be performed. Workbook 3 discusses a number of JOVIAL (J73) executable statements, beginning with this brief review of the assignment-statement.

An assignment-statement causes the value to the right of the equal sign (the source value) to be assigned to the variable(s) to the left of the equal sign (the target(s)). In performing the assignment, the source is evaluated first. Then, the leftmost target variable is evaluated and the value of the source is assigned. This process continues until all target variables have been assigned the source value.

> *NOTE: The type of the source must be equivalent or implicitly convertible to the type of the target variable(s).*

REVIEW EXERCISES

Declarati s

ITEM AA S 7;

ITEM BB F 20;

ITEM CC C 4;

What must the type of XX be in each of the following assignments?

Assignment-statements	Type of XX
AA = XX;	
BB = XX;	
CC = XX;	
XX = BB;	
XX,AA = -49;	
XX,BB = -6.2E7;	
AA = XX + -79;	

ANSWERS

Declarations

ITEM AA S 7;

ITEM BB F 20;

ITEM CC C 4;

What must the type of XX be in each of the following assignments?

Assignment-statements	Type of XX
AA = XX;	any S, U
BB = XX;	F 20 or smaller
CC = XX;	any C
XX = BB;	F 20 or greater
XX,AA = -49;	any S, U
XX,BB = -6.2E7;	any F
AA = XX + -79;	any S, U

SECTION 2

CONDITIONAL STATEMENTS

THE IF-STATEMENT

The if-statement provides for conditional execution of a statement depending upon the value of its Boolean formula. This value is often the result of a relational expression and is, of course, either true or false. The general form of the if-statement is

```
IF  test;
        true-alternative;
ELSE
        false-alternative;
```

If the result of test is TRUE, the true-alternative is executed. The false-alternative is ignored, and processing continues at the next statement.

If the result of test is FALSE, the true-alternative is ignored and the false-alternative is executed.

Processing continues at the next statement.

Example

```
IF SUM > LIMIT;
        SUM = SUM/2;
ELSE
        SUM = SUM+1;
ANSWER = SUM;
```

RELATIONAL EXPRESSIONS

While test can be set to TRUE or FALSE from the outset, this value is more frequently obtained by the evaluation of a relational expression.

A relational expression is an expression with a relational operator.

The relational operators are:

= <> < >= < <=

The operands of a relational expression must be equivalent or implicitly convertible.

The operands that may use all the above operators are:

S, U, F, A, C, or STATUS.

Bit types may only use the relational operators equal (=) and not equals (<>).

The result of a relational expression is a Boolean value:

TRUE (1B'1') or FALSE (1B'0')

<u>Examples</u>

Given the following declarations:

```
TYPE MASK B 1;

TYPE NUM S 4;

TYPE LETTER STATUS (V(A), V(B), V(C));

TYPE SEQ'QF'B STATUS (V(B), V(BB), V(BBB));

ITEM AA MASK = TRUE;

ITEM BB NUM = -6;

ITEM CC S 7 = 47;

ITEM DD LETTER = V(A);

ITEM EE B 1 = 1B'1';

ITEM FF SEQ'OF'B = V(BBB);
```

The following are valid relational expressions:

Expression	Result
AA <> EE	F
FF > V(B)	T (V(B) is disambiguated)
BB <= CC	T
V(B) = DD	F (V(B) is disambiguated)

NESTED IF-STATEMENTS

If-statements may be nested. For example,

```
IF SUM > LIMIT;
        IF READY;
                SUM = SUM/2;
        ELSE
                IF CHARGED;
                          SUM = -SUM;
                ELSE
                          SUM = SUM-9;
ELSE
        SUM = SUM+1;
ANSWER = SUM;
```

When if-statements are nested, the else-clause, if present, is associated with the innermost if that does not already have an else-clause.

Example

```
IF MASK = CODE; IF FINAL = LIMIT; TOTAL = 100; ELSE
TOTAL = 0;
```

Formatted to show the association of the else-clause:

```
IF MASK = CODE;
        IF FINAL = LIMIT;
                TOTAL = 100;
        ELSE
                TOTAL = 0;
```

To associate a dangling else-clause with an if-clause other than the default association, a null-statement (;) or a compound-statement (discussed below) may be used. The null-statement fulfills the requirement for a statement, but does not perform any action.

Examples

```
IF MASK = CODE;
        BEGIN (compound-statement)
        IF FINAL = LIMIT;
                TOTAL = 100;
        END
ELSE
        TOTAL = 0;
IF MASK = CODE;
        IF FINAL = LIMIT;
                TOTAL = 100;
        ELSE
                ; (null-statement)
ELSE
        TOTAL = 0;
```

MISSING ELSE-CLAUSES

An if-statement may be written without an else-clause. The form is:

```
IF test ;
        true-alternative ;
```

If the result of test is TRUE, the true-alternative is executed. Processing continues at the next statement.

If the result of test is FALSE, the true-alternative is ignored and processing continues at the next statement.

Example

```
IF ALPHA <> BETA;
        ALPHA = GAMMA;
BETA = GAMMA;
```

COMPOUND-STATEMENTS

The statements discussed thus far (assignment, if and null) are simple-statements. A compound-statement is a sequence of simple-statements grouped together to be treated as a simple-statement. The form is

```
BEGIN
simple-statement ...
END
```

> *NOTE: A special form of a null-statement is:*
> *BEGIN*
> *END*

Example

```
IF READY;
        BEGIN
        SUM = SUM/2;
        READY = FALSE;
        COUNT = COUNT + 1;
        END
ELSE
        BEGIN
        SUM = -SUM;
        READY = TRUE;
        COUNT = COUNT -1;
        END
```

LABELS

Labels may be attached to statements. A label is a name followed by a colon. The form of a label is

name :

Any number of labels may be placed immediately before a simple-statement as follows:

[label ...] simple-statement

Any number of labels may be placed before the BEGIN or END of a compound-statement as follows:

```
[ label ... ]  BEGIN
               simple-statement ..
[ label ...]   END
```

Example

```
ONE :           COUNT = COUNT + 1;
                  IF READY;
TWO : THREE:        BEGIN
                    SUM = COUNT;
                    ANSWER = TRUE;
                    END
                ELSE
                    COUNT = 0;
```

GOTO-STATEMENT

A goto-statement is often used to branch upon a particular condition, though it can be used independently of an if-statement. A goto-statement transfers control to the statement labeled by the given statement name. The form is

```
GOTO  statement-name;
```

Example

```
    IF LIMIT/2 = 4;
          BEGIN
          .
          .
          .
          IF SUM = 0;
                     GOTO L'ONE;
          .
          .
          .
          END
    ELSE
          SUM = LIMIT;
L'ONE : ANSWER = SUM;
```

> *NOTE: Statement-name must be known within the scope in which the goto appears. Statement-name must not be the label of a statement within the controlled statement of a loop unless the goto is also within the same controlled statement. Statement-name must be the name of a statement that is in an enclosing subroutine or in another module.*

COMPILE-TIME-KNOWN TESTS

An if-statement with a compile-time-known test is resolved by the compiler and reduced to a single alternative.

Example

```
CONSTANT ITEM READY B 1 = TRUE;
.
.
.
IF READY;
        SUM = SUM/2;
ELSE
        SUM = SUM-9;
ANSWER = SUM;
```

Because READY will always be TRUE in this example, the compiler generates code only for the true-alternative of the if-statement.

```
SUM = SUM/2;
ANSWER = SUM;
```

> *NOTE: All unselected alternatives must still be syntactically correct.*

SHORT-CIRCUITING

In some cases the value of a logical formula is known before all the parts of the formula are evaluated. When the value of the formula is known, further evaluations are 'short-circuited'.

Example

```
ITEM SUM S;

ITEM LIMIT S;

ITEM TEST'BIT B 1;

IF TEST'BIT            AND        LIMIT > SUM ;

evaluated first,

if FALSE, no
further evaluation
done, the result
of the test
is FALSE

if TRUE,  ------------------->    evaluated second,
                                  result of this
                                  relational expression
                                  becomes the result of
                                  if test
```

If both operands have a size of one bit and the value of the left operand is such that the result of the operator can be determined, the right operand will not be evaluated. (This rule works only with the operators AND, OR.)

IF-STATEMENTS -- EXERCISES

What is "wrong" with the following if-statements? Make the necessary corrections.

①
```
IF NAME = NUM;
        COUNT = COUNT +1;
ELSE
        NAME = NAME +1;
        COUNT = 0;
```

②
```
IF BOOKS + MAGS < 100;
        IF LIBRARY = 'CENTRAL';
                COUNT = COUNT +1;
ELSE
        BOOKS = 40;
```

③
```
ITEM PROFIT F;
ITEM EXPENSE A 10,4;
IF EXPENSE < PROFIT OR PROFIT > = 100;
        BALANCE = PROFIT - EXPENSE;
        GOOD'YEAR = TRUE;
ELSE
        GOOD'YEAR = FALSE;
```

ANSWERS

What is "wrong" with the following if-statements? Make the necessary corrections.

①
```
IF NAME = NUM;
        COUNT = COUNT +1;
ELSE
        NAME = NAME +1;
        COUNT = 0;
```

indent implies it's associated with false-alternative; treated syntactically as statement after IF-ELSE

②
```
IF BOOKS + MAGS < 100;
        IF LIBRARY = 'CENTRAL';
                COUNT = COUNT +1;
ELSE
        BOOKS = 40;
```

indent implies it's associated with outer IF; treated syntactically as associated with inner IF

③
```
ITEM PROFIT F;
ITEM EXPENSE A 10,4;
IF EXPENSE < PROFIT OR PROFIT > = 100;
        BALANCE = PROFIT - EXPENSE;
        GOOD'YEAR = TRUE;
ELSE
        GOOD'YEAR = FALSE;
```

missing BEGIN
END

SECTION 3

CASE-STATEMENTS

SOFTECH

CASE-STATEMENT

Whereas an if-statement provides for the conditional execution of either of two statements, a case-statement provides for a choice of executing one or more of a number of statements.

Case-statements eliminate many of the problems associated with multiple nested IF-statements. For example, consider the following:

```
IF NUMBER < 0 OR NUMBER > = 12;

        COUNT = 0;

ELSE

        IF NUMBER = 1;

                COUNT = COUNT + 1;

        ELSE

                IF NUMBER = 2 OR NUMBER = 3;

                        COUNT = COUNT + 2;

                ELSE

                        IF (NUMBER > = 4 and NUMBER < - 7) OR
                        NUMBER = 11;

                                COUNT = COUNT + 3;

                        ELSE

                                IF NUMBER = 8 OR NUBMER = 10;

                                        COUNT = COUNT + 4;

                                ELSE

                                        COUNT = COUNT + 5;
```

At first glance, the logic behind these nested ifs is a bit confusing. Now look at the same actions specified by a case-statement:

CASE NUMBER;		CASE case-selector ;
BEGIN		BEGIN
(DEFAULT) :	COUNT = 0;	[default-option]
(1) :	COUNT = COUNT + 1;	
(2,3) :	COUNT = COUNT + 2;	
(4:7,11) :	COUNT = COUNT + 3;	case-option ...
(8,10) :	COUNT = COUNT + 4;	
(9) :	COUNT = COUNT + 5;	
END		END

The form of a simple case-option is:
(case-index) : statement

The syntax of the case-statement takes care of *all the explicit* if tests internally.

In this case-statement, the case-selector is matched with a case-index, one action is performed, and processing continues after the END of the case-statement.

The syntax of the case-statement is

```
CASE  case-selector ;
      BEGIN
      [default option]
      case option ...
      END
```

A case-option is of the form

(case-index) : statement

case-selector The case-selector is the variable name the programmer wishes to test. The case-selector may be of type S, U, B, C, or STATUS only. The types of all the case-indices must be equivalent or implicitly convertible to the type of the case-selector. If the case-selector does not match any of its case-options the default option is selected. In such a case, if no default option is given, the program is invalid.

case-index A case-index may be a single value, a pair of bounds or any combination of the two. The form of a pair of bounds is

lower-bound : upper-bound

> *NOTE: A pair of bounds may be used only if the case-index (and case-selector) are of type S, U, or STATUS with default representations.*

The case-index must be known at compile-time and be of type S, U, B, C or STATUS only. The type of the case-index must be equivalent or implicitly convertible to the type of the case-selector.

A case selector matches a case-index if it is either equal to the case-index or lies within the bounds of the case-index. All case-indices must be distinct.

FALLTHRU OPTION

A FALLTHRU may be given at the end of a default-option or a case-option. The form is:

(DEFAULT) : statement [FALLTHRU]

(case-index) : statement [FALLTHRU]

An option is selected, its statement is executed, if a FALLTHRU is encountered the following statement is executed, if a FALLTHRU is encountered the following statement is executed, ... until no FALLTHRUs are encountered. Processing then continues after the END of the case-statement.

FALLTHRU IN A CASE-STATEMENT -- EXERCISE

```
SALES, COUNT = 0;

CASE PRICE;
       BEGIN

       (DEFAULT):      SALES = 100;

       ('A','B');      BEGIN
                       SALES = 1;
                       COUNT = COUNT + 1;
                       END FALLTHRU

       ('C'):          SALES = SALES + 3; FALLTHRU

       ('D', 'E', 'F'): BEGIN
                       SALES = SALES + 5;
                       COUNT = COUNT + 2;
                       END

       ('G'):          SALES = SALES + 7;

       ('H', 'I'):     BEGIN
                       SALES = SALES + 9;
                       COUNT = COUNT + 1;
                       END FALLTHRU

       ('J'):          SALES = SALES + 11;

       END
```

Given the following initial values of PRICE find the value of SALES and COUNT at the end of the case-statement.

Value of PRICE	Value of Sales	Value of COUNT
PRICE = 'B'		
PRICE = 'C'		
PRICE = 'F'		
PRICE = 'G'		
PRICE = 'H'		
PRICE = 'J'		
PRICE = 'K'		

ANSWERS

```
SALES, COUNT = 0;

CASE PRICE;
      BEGIN

      (DEFAULT):      SALES = 100;

      ('A','B');      BEGIN
                      SALES = 1;
                      COUNT = COUNT + 1;
                      END FALLTHRU

      ('C'):          SALES = SALES + 3; FALLTHRU

      ('D', 'E', 'F'): BEGIN
                      SALES = SALES + 5;
                      COUNT = COUNT + 2;
                      END

      ('G'):          SALES = SALES + 7;

      ('H', 'I'):     BEGIN
                      SALES = SALES + 9;
                      COUNT = COUNT + 1;
                      END FALLTHRU

      ('J'):          SALES = SALES + 11;

      END
```

Given the following initial values of PRICE find the value of SALES and COUNT at the end of the case-statement.

Value of PRICE	Value of Sales	Value of COUNT
PRICE = 'B'	9	3
PRICE = 'C'	8	2
PRICE = 'F'	5	2
PRICE = 'G'	7	0
PRICE = 'H'	20	1
PRICE = 'J'	11	0
PRICE = 'K'	100	0

COMPILE-TIME KNOWN CASE-STATEMENTS

If the case-selector in a case-statement is known at compile-time, the case-statement is reduced to the statement identified by the matching case-index and all statements to which the selected statement falls through.

Example

```
CONSTANT ITEM MONTH STATUS (V(JA), V(FE), V(MR), V(AP),
V(MY), V(JN), V(JY), V(AG), V(SP), V(OC), V(NV), V(DC)) =
V(FE);

CASE MONTH;
     BEGIN
     (V(JA) : V(MR)):     BEGIN
                          PROFITS = SALES * PRICE;
                          MAKE'REPORT = TRUE;
                          END FALLTHRU

     (V(AP) : (V(JN)):    PAYROLL = FALSE;

     (V(JY) : V(SP)):     BEGIN
                          PROFITS = SALES * NEW'PRICE;
                          MAKE'REPORT = TRUE;
                          END FALLTHRU

     (V(OC) : V(DC));     PAYROLL = TRUE:

     END
```

The compiler generates code for the following:

```
PROFITS = SALES * PRICE;

MAKE'REPORT = TRUE;

PAYROLL = FALSE;
```

> *NOTE: All unselected alternatives must be syntactically correct.*

CASE-STATEMENTS -- EXERCISES

The following case-statements are all incorrect. Mark the incorrect parts:

1.

```
CASE SPEED;
      BEGIN
      (1:10):          ACTION1;
      (2:4):           ACTION2;
      (12,25:100):     ACTION3;
      END
```

2.

```
CASE INDEX;
      BEGIN
      (1:10):          ACTION1;
      (11:100):        ACTION2;
      (DEFAULT):       ERROR9;
      END
```

3.

```
CASE ALPHA;
      BEGIN
      (2,4,6,8,10):    ACTION1 FALLTHRU;
      (1,3,5,7,9):     ACTION2;
```

4.

```
CASE RADIUS;
      (DEFAULT):       ACTION1;
      (1:100):         ACTION2;
```

ANSWERS

The following case-statements are all incorrect. Mark the incorrect parts:

1.
```
CASE SPEED;
        BEGIN
        (1:10):        ACTION1;
        (2:4):         ACTION2;  <----------- case-index not
        (12,25:100):   ACTION3;               distinct
        END
```

2.
```
CASE INDEX;
        BEGIN
        (1:10):        ACTION1;
        (11:100):      ACTION2;
        (DEFAULT):     ERROR9;   <------------default should
        END                                   appear first
```

3.
```
CASE ALPHA;
        BEGIN
        (2,4,6,8,10):  ACTION1 FALLTHRU; <--- semi-colon should
        (1,3,5,7,9):   ACTION2;               appear before
                                              FALLTHRU
```

4.
```
CASE RADIUS;
        (DEFAULT):     ACTION1; <---------    case-statement
        (1:100):       ACTION2;          >    needs BEGIN,
                                <--------     END
```

SECTION 4

LOOP-STATEMENTS

LOOP-STATEMENT

A loop statement repeatedly executes a controlled-statement. There are two general types of loops in JOVIAL (J73), the while-loop and the for-loop. These may be combined to give the programmer added control.

WHILE-LOOPS

One kind of loop-statement is a while-loop. A while-loop is a simple-statement. A while-loop is of the form

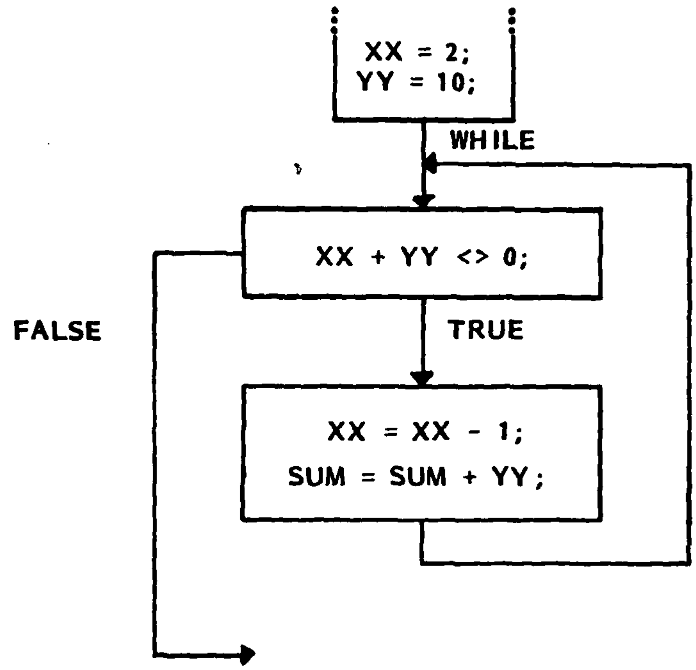
```
WHILE  condition;
       controlled-statement
```

Condition is a bit formula whose size is one bit. For instance, condition could be the result of a relational expression is of type B1 - either TRUE or FALSE.

Example

```
XX = 2;
YY = 10;
WHILE XX + YY <> 0;
       BEGIN
       XX = XX - 1;
       SUM = SUM + YY;
       END
```

The flow of control of this WHILE loop can be diagrammed as follows:



The statements within the loop will continue to execute as long as the sum of XX and YY is not equal to zero.

<u>FOR-LOOPS</u>

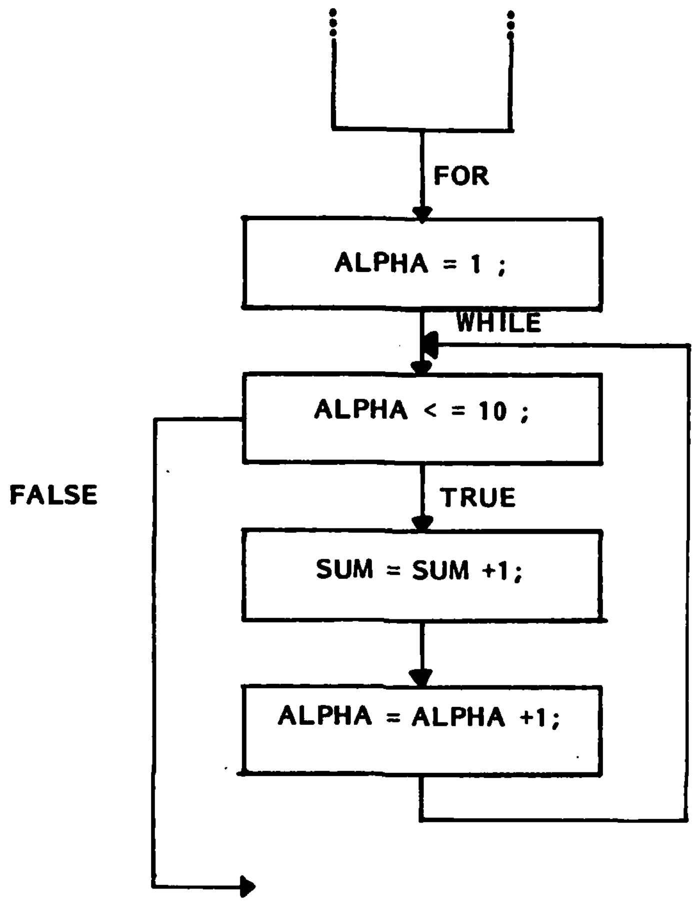
The other type of loop is the for-loop. The general form of a for-loop is:

FOR loop-control : *initial value* { BY increment / THEN next-value } WHILE condition;

<u>Example</u>

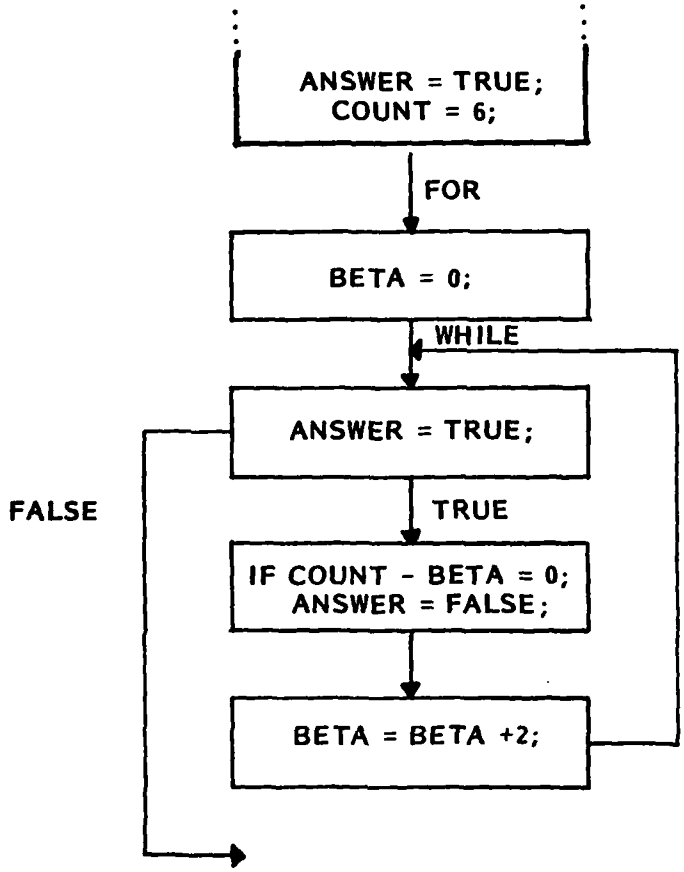
```
FOR ALPHA  :  1  BY  1  WHILE  ALPHA  <  =  10;
        SUM = SUM + 1;
```

The flow of control for this loop can be diagrammed as follows:



As a further example, consider the following:

```
ANSWER = TRUE;
COUNT = 6;
FOR BETA: 0 BY 2 WHILE ANSWER = TRUE;
        IF COUNT - BETA = 0;
                ANSWER = FALSE;
```



loop-control The loop control variable may be either a previously declared variable or a single letter. At all times, the type of the initial-value and the type of the increment or next-value must be equivalent or implicitly convertible to the type of loop-control.

There are special restrictions on a single letter loop control. These are as follows:

- No item-declaration is needed; the single letter loop-control is implicitly declared by its use.
- The single letter loop-control is not known outside the scope of the loop.

- The type of a single letter loop-control is the type of its initial-value.

- The initial value of a single letter loop-control must not be anambiguous status-constant.

- The value of a single letter loop-control must not be altered within the loop, (i.e., a single letter loop-control must not be the target of an assignment-statement.

Examples

This loop is incorrect

```
FOR I : 1 BY 1 WHILE < 10;

        I = COUNT + TOTAL;
```

The single letter loop-control must not be the target in an assignment.

This loop is correct:

```
FOR I : 1 BY 1 WHILE I < 10;

        COUNT = 1 + TOTAL;
```

The single letter loop-control may be used as a part of a formula.

This loop is incorrect:

```
FOR I : 1 BY 2 WHILE I < = 100;

        BEGIN

        SUM = 0;

        FOR J : 1 BY 1 WHILE J < I;

                SUM = SUM + J;

        TOTAL = SUM / J;

        END
```

The single letter loop-control (J) must not be used outside of the loop it controls.

BY-clause The by-clause specifies an increment for the loop control variable. This increment is added to the loop control variable after each iteration of the loop. The increment can be any numeric formula.

THEN-clause The value of loop-control may be changed by repeated assignment by use of a then-clause. The form is:

```
FOR loop-control : initial value THEN next-value

WHILE condition ;

        controlled-statement
```

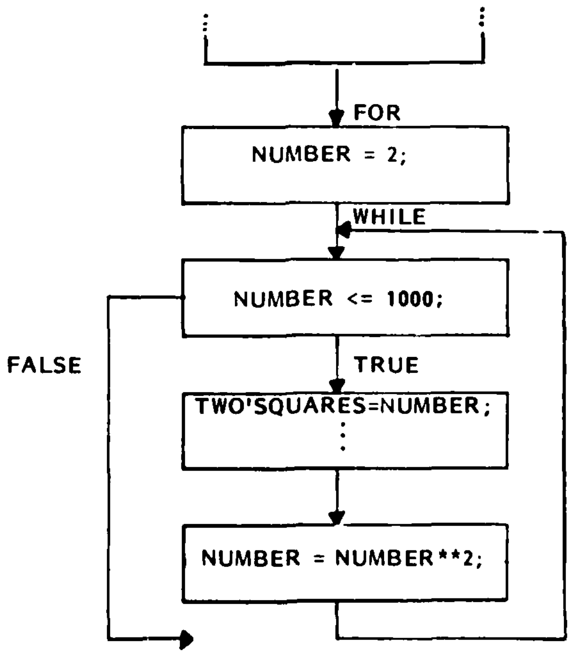
Example

```
FOR NUMBER:  2 THEN NUMBER**2 WHILE NUMBER < = 1000;

        BEGIN

        TWO'SQUARES = NUMBER;
        .
        .
        .
        END
```

The flow of control can be diagrammed as follows:



NESTED LOOP-STATEMENTS

A loop may be nested within other loop-statements.

Examples

```
FOR XX : 0 BY 1 WHILE XX < = 10 AND (NOT TEST'BIT);
        FOR YY: 10 BY -1 WHILE YY < = 5 AND (NOT TEST'BIT);
                FOR ZZ: 0 BY 4 WHILE ZZ < = 100 AND (NOT TEST'BIT)
                        IF 5 = XX*3 + YY*9 + ZZ*(-4);
                                TEST'BIT = TRUE;
```

The above example finds the first solution to the equation:

$3X + 9Y - 4Z = 5$

with allowable values of X being 0 through 10, allowable values of Y being 10 through 5, and allowable values of Z being multiples of 4 from 0 through 100.

EXIT STATEMENT

An exit-statement is used to effect an exit from a loop. The form is:

```
EXIT ;
```

Example

```
FOR INDEX: SOME'VALUE THEN INDEX/2 WHILE INDEX > 0;
        BEGIN
        IF INDEX MOD 5 = 0;
                EXIT;
        COUNT = COUNT + 1;
        END
```

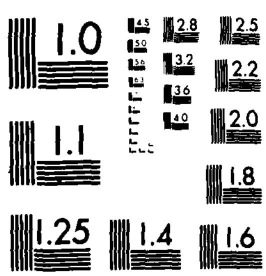
AD-A108 527 SOFTECH INC WALTHAM MA F/G 5/9
THE JOVIAL (J73) WORKBOOK. VOLUME I. INTEGER AND FLOATING POINT--ETC(U)
NOV 81 F30602-79-C-0040
UNCLASSIFIED RADC-TR-81-333-VOL-1 NL

3 of 5
AD
A108527

SIFIED

3 OF 3

AD
A108527



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

When the if-statement is evaluated and found to be TRUE, the exit-statement is executed, COUNT = COUNT + 1 is not executed, and processing continues after the END of the for-loop.

SECTION 5

SUMMARY

SOFTECH

EXECUTABLE STATEMENTS

An executable statement specifies an action to be performed.

The executable statements shown so far are:

assignment-statement

if-statement

goto-statement

case-statement

loop-statement

exit-statement

A null-statement has the form:

;

A compound-statement has the form:

BEGIN

simple-statement ...

END

An assignment-statement assigns a source (value) to one or more targets (variables). The form is:

target, ... =source ;

An exit-statement is used to exit from a loop. The form is:

EXIT ;

LABELS, GOTO-STATEMENT

A goto-statement transfers control to the statement labelled by the given statement-name. The form of a label is:

name :

The form of a goto-statement is:

GOTO statement-name;

> *NOTE: Control may not be sent to a label within a controlled-statement from outside the loop. Labels within a loop are only accessible from a goto-statement within that loop.*

IF-STATEMENTS

The general form of an if-statement is:

IF test ;

true-alternative

[ELSE false-alternative]

Test is a Boolean value. For example: a relational expression, any bit string, (it will be implicitly converted to a B 1, the right-most bit), or a formula of any type explicitly converted to a bit string.

If-statements may be nested; a dangling else-clause is associated with the inner most if-statement without an else-clause.

If the value of test is known at compile-time, the compiler will only generate code for the selected alternative.

The evaluation of a logical formula may be 'short-circuited': if the value of the whole formula can be determined before the evaluation of all the parts, evaluation will stop.

CASE-STATEMENTS

A case-statement is a special form of an if-statement.

The general form of a case-statement is:

```
CASE  case-selector  ;
      BEGIN
      [ default-option ]
      case-option ...
      END
```

The case-selector is the variable name the programmer wishes to test. The case-selector may be of type S, U, B, C, or STATUS. If the case-selector does not match any of the case-options, the default-option is selected. In such a case, if no default-option is given, the program is invalid.

A default-option has the form:

```
( DEFAULT )  :  statement  [ FALLTHRU ]
```

A case-option has the form:

```
( case-index ) : statement [ FALLTHRU ]
```

The type of the case-index must be equivalent or implicitly convertible to the type of the case-selector. A case-index may be a value or a range of values or any combination of the two. A range may be given only if the case-selector is of type S, U, or STATUS. The case-index must be known at compile-time and must be distinct from case-option to case-option.

A FALLTHRU at the end of a default-option or a case-option allows processing to fall through to the next option.

If the value of case-selector is known at compile-time, the compiler will only generate code for the selected option, (options, should the selected option have a FALLTHRU).

LOOP-STATEMENTS

A loop-statement repeatedly executes a controlled-statement.

JOVIAL (J73) has two kinds of loop-statements:

while-loop

for-loop

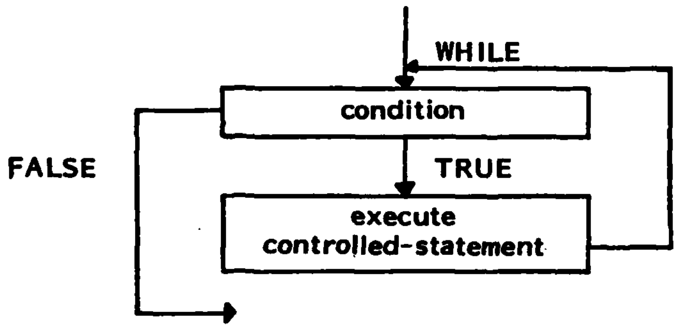
A while-loop is of the form:

WHILE condition ;

controlled-statement

Condition is a Boolean value. For example: a relational expression, any bit string, (it will be implicitly converted to a B 1, the right-most bit), or a formula of any type explicitly converted to a bit string.

The flow of control can be diagrammed:



FOR-LOOPS

The general form of a for-loop is:

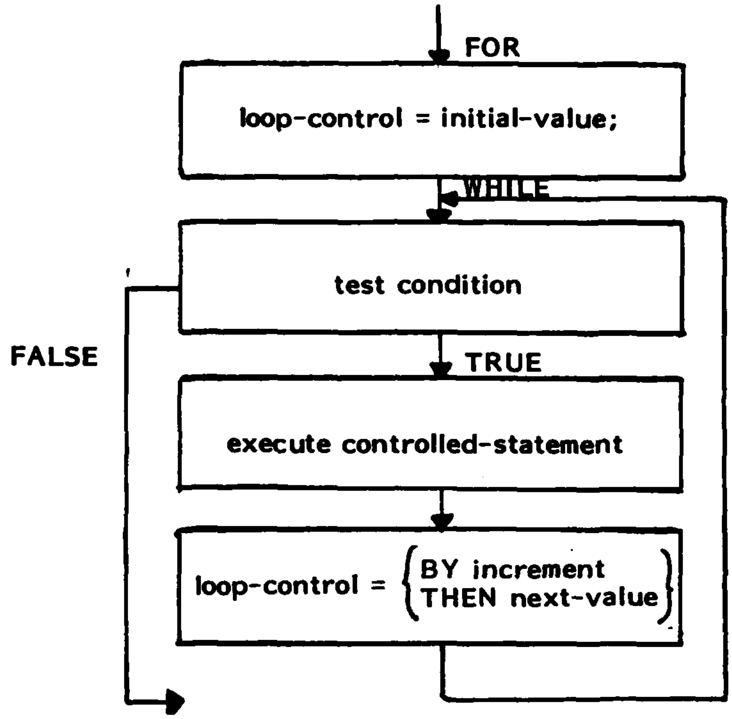
FOR loop-control : initial-value [{ BY increment / THEN next-value }]
[WHILE condition];

controlled-statement

or

FOR loop-control : initial value [WHILE condition] [$\left\{\begin{array}{l}\text{BY increment}\\ \text{THEN next-value}\end{array}\right\}$];

The flow of control can be diagrammed:



The by-clause gives the increment to be added to the loop-control each time the controlled-statement is executed. The type of increment must be equivalent or implicitly convertible to the type of loop-control. Increment must be a numeric formula.

The then-clause gives the new value to be assigned to the loop-control each time the controlled-statement is executed. The type of next-value must be equivalent or implicitly convertible to the type of loop-control. Next-value may be a formula of any type.

Loop-control may be either a previously declared item or a single letter.

At all times, the type of increment or next-value must be equivalent or implicitly convertible to the type of loop-control.

Special restrictions on single letter loop-control are:

- No item-declaration needed.
- Type of single letter loop-control is type of initial-value.
- Value of single letter loop-control NOT known outside of loop.
- Value of single letter loop-control must NOT be altered within controlled-statement.

THE JOVIAL (J73) WORKBOOK

VOLUME 4

TABLE AND BLOCK DECLARATIONS

1081-1

April 1981

Submitted to

Department of the Air Force
Rome Air Development Center
I S I S
Griffiss Air Force Base, NY 13441

Prepared by

SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02154

PREFACE

Workbook 4 is intended for use with Tape 4 of the JOVIAL (J73) Video Course. Its purpose is to elaborate upon and reinforce concepts and language rules introduced in the videotape.

This workbook discusses in three sections non-dimensioned tables, dimensioned tables and blocks. Specifically addressed topics in sections one and two are presets, constant tables, typed tables and the like-option. Section 2 also contains information regarding dimensions, subscripts and type matching. Section 3 discusses blocks. Section 4 is a summary of the material presented in this segment.

TABLE OF CONTENTS

SYNTAX

The syntax conventions used in the JOVIAL (J73) Video Course and workbook are as follows:

<table>
<tr><th>Syntax</th><th>Meaning</th><th>Sample Expansions</th></tr>
<tr><td>[some-feature]</td><td>Brackets indicate an optional feature.</td><td>some-feature
OR
nothing</td></tr>
<tr><td>{one | other}</td><td>Braces with a vertical bar indicate disjunction- a choice between alternatives.</td><td>one
OR
other</td></tr>
<tr><td>{this-one
that-one}</td><td>Braces with each feature on separate lines indicate disjunction - a choice between alternatives.</td><td>this-one
OR
that-one</td></tr>
<tr><td>letter ...</td><td>The sequence '...' indicates one or more repetitions of a feature.</td><td>letter
letter letter
letter letter letter
...</td></tr>
<tr><td>(letter),...</td><td>The sequence "..." following a comma (or a colon) indicates one or more repetitions of the feature separated by commas (or colons).</td><td>(letter)
(letter) (letter)
(letter) (letter) (letter)
...</td></tr>
<tr><td>[{this-one
that-one}]
+ another</td><td>Syntax symbols may be combined.</td><td>this-one + (another)
that-one + (another)
+ (another)</td></tr>
</table>

SECTION 1

SIMPLE TABLES

TABLES

Tables are collections of values gathered together to form a single data object. They are used for the constructs called "records" and "arrays" in other languages. A simple or non-dimensional table is like a record. A simple table-declaration is of the form:

```
TABLE name;

        entry-description;
```

TABLE JOVIAL (J73) reserved word used to declare a data object of type TABLE. It has a specific meaning to the compiler, which restricts its use to declarations. It cannot be used as a name.

name A name is a sequence of letters, digits, dollar signs and primes (single quote characters). It must begin with a letter or a dollar sign, and be at least two characters long. A name may be any length, though only the first 31 characters are examined for uniqueness within the program. (Fewer than 31 characters may be examined for external names. External names are discussed in Workbook 8).[1]

entry-description An entry-description is the declaration of the scalar components of a table. An entry-description may be simple (one entry consisting of one item only) or compound (a number of items in one entry).

Examples:

1.
```
TABLE ID;

        ITEM NAME C 20;
```

Table ID is a simple (non-dimensioned) table with a simple entry-description (i.e., one item). Table ID has one entry. That one entry is composed of one item -- NAME.

[1]See list of JOVIAL (J73) reserved words given in Workbook 1, page 3-2.

2. TABLE FULL'ID;

```
BEGIN
ITEM NAME C 20;
ITEM SOC'SEC'NO C 9;
ITEM AGE U 7;
ITEM GRAD'HS B:
END
```

Table FULL'ID is a simple (non-dimensioned) table with a compound entry-description. FULL'ID has one entry. That one entry is composed of four items - NAME, SOC'SEC'NO, AGE, GRAD'HS.

TABLE-PRESETS

A table-preset initializes items within an entry of a table (see Figure 1-2).

A table-preset may be given as a part of the entry-description or as part of the table-heading -- not both.

Example

```
TABLE FULL'ID;
    BEGIN
    ITEM NAME  C 20 = 'MR.X';
    ITEM SOC'SEC'NO C 9 = '111559999;'
    ITEM AGE U 7 = 42;
    ITEM GRAD'HS B = TRUE;
    END
```

OR

```
TABLE FULL'ID = 'MR.X', '111559999', 42, TRUE;
    BEGIN
```

```
        ITEM NAME C 20;
        ITEM SOC'SEC'NO C 9;
        ITEM AGE U 7;
        ITEM GRAD'HS B;
        END
```

Both forms of the table-preset have the same effect. The type of the preset values must be either equivalent or implicitly convertible to the type of the items being preset.

An omitted value in a table-preset indicates that the corresponding item is not initialized. Items not preset have unknown value.

Example

```
TABLE FULL'ID = 'MR.X', , , TRUE;
        BEGIN
        ITEM NAME C 20;
        ITEM SOC'SEC'NO C 9;
        ITEM AGE U 7;
        ITEM GRAD'HS B;
        END
```

In this example, only the items NAME and GRAD'HS are preset. SOC'SEC'NO and AGE have unknown value.

Example

```
TABLE FULL'ID = 'MR.X' , '111559999'
        BEGIN
        ITEM NAME C 20;
        ITEM SOC'SEC'NO C 9;
        ITEM AGE U 7;
        ITEM GRAD'HS B;
        END
```

The tables shown so far could be diagrammed in the following way:

```
TABLE DIMENSIONS;
      BEGIN
      ITEM LENGTH U;
      ITEM WIDTH U;
      ITEM HEIGHT U;
      END
```

LENGTH	WIDTH	HEIGHT

DIMENSIONS

Given the following table-presets, the following diagrams would result:

TABLE DIMENSIONS
= 1,2,3;

LENGTH = 1	WIDTH = 2	HEIGHT = 3

TABLE DIMENSIONS
= 2,,6;

LENGTH = 2	WIDTH	HEIGHT = 6

TABLE DIMENSIONS
=3(10);

LENGTH = 10	WIDTH = 10	HEIGHT = 10

Notes

The table-presets given here as a part of the table-heading could have been given as a part of the entry-descriptions.

Figure 2-1. Table Presets

In this example, only the items NAME and SOC'SEC'NO are preset. AGE and GRAD'HS have unknown value.

A repetition-count may be used in a table-preset to repeat a value or a set of values. The form is:

```
repetition  ( preset-option, ...)
```

A preset-option is of the form:

$$\left\{\begin{array}{ll}\text{value} & \\ \text{repetition} & (\text{ preset-option, ... })\end{array}\right\}$$

Given the following table-declarations:

```
TABLE SALES;
    BEGIN
    ITEM SUN U;
    ITEM MON U;
    ITEM TUE U;
    ITEM WED U;
    ITEM THU U;
    ITEM FRI U;
    ITEM SAT U;
    END
```

The following table-presets have the following effects:

TABLE SALES = 7 (0);	All items are preset to 0.
TABLE SALES = 1, 3 (0,1);	SUN is preset to 1, MON to 9, TUE to 1, WED to 0, THU to 1, FRI to 0, SAT to 1.
TABLE SALES = 2(2(5),3);	SUN is preset to 5, MON to 5, TUE to 3, WED to 5, THU to 5, FRI to 3, SAT is not preset.

> *NOTE: While nested repetition-counts are allowed in the language, it is not recommended that they be used because or their drain on memory.*

CONSTANT TABLES

A table may be declared to be constant, in which case the value(s) of the entry description may not be altered during program execution. The form is:

```
CONSTANT  TABLE  name  [table-preset];

        entry-description  [table-preset];
```

The constant-table preset may be given either as part of the table-heading or as part of the entry-description (not both).

Example

```
CONSTANT TABLE LOOK'UP = 10.0, 100.0, 1000.0, .1;

        BEGIN

        ITEM NUMBER F;

        ITEM SQR F;

        ITEM CUBE F;

        ITEM RECIP F;

        END
```

All items within a constant table do n t need to be preset, though these items will have constant unknown value during execution.

TABLE TYPE-DECLARATIONS

A table type-declaration can be used to declare a descriptive template for tables which can be referred to by name. This table-type-name may then be used in any number of table-declarations to define a table with the attributes of that typed table. The form of a simple table type-declaration is:

```
TYPE  table-type-name  TABLE;
        entry description;
```

> *NOTE: Since a type-declaration declares no data it cannot have a preset.*

Once a table type-declaration has been made, the table type-name associated with that template may be used in a table declaration:

```
[CONSTANT]  TABLE  name  table-type-name  [preset];
```

For example, given the following table type-declaration:

```
TYPE MAP TABLE;
        BEGIN
        ITEM LONGITUDE F 20;
        ITEM LATITUDE S 15;
        END
```

The following table-declarations show correct uses of the table type-name MAP:

```
TABLE USA MAP ;
TABLE GLOBE MAP;
```

In this case, the table USA "looks like" type MAP; it has one entry with two items, LONGITUDE of type F 20, and LATITUDE of type S 15. The table GLOBE "looks like" MAP; it, too, has one entry with two items, LONGITUDE of type F 20, and LATITUDE of type S 15.

> *NOTE: Pointers must be used to reference items in tables declared using a table-type-name. Pointers will be discussed in Workbook 5.*

LIKE-OPTION

A table type-declaration may include a like-option. A like-option allows the programmer to create several variable length records from a common beginning. The form is:

```
TYPE table-type-name TABLE LIKE table-type-name;
     entry description;
```

Examples:

```
TYPE SHORT'ID TABLE;
     BEGIN
     ITEM NAME C 20;
     ITEM ADDRESS C 50;
     END
TYPE A'MED'ID TABLE LIKE SHORT'ID;
     BEGIN
     ITEM PHONE C 10;
     ITEM MARRIED B;
     END
```

```
TYPE B'MED'ID TABLE LIKE SHORT'ID;
        BEGIN
        ITEM BIRTH C 6;
        ITEM EDUC STATUS (V(GRADE), V(HS), V(BACH));
        ITEM EMPLOYED B;
        END
```

These declarations could be diagrammed as follows:

SHORT'ID

NAME	ADDRESS

A'MED'ID

NAME	ADDRESS	PHONE	MARRIED

B'MED'ID

NAME	ADDRESS	BIRTH	EDUC	EMPLOYED

Once these templates have been established, any number of tables could be declared using the type-name associated with a given template --

```
TABLE REGISTER SHORT'ID;
TABLE FONE'BOOK A'MED'ID;
TABLE SURVEY B'MED'ID;
```

(See Section 2, page 28 for a discussion of type equivalence and conversion.) w/c

UNNAMED ENTRY-DESCRIPTION

One additional form of the table-declaration is allowed, an unnamed entry-description. The form is:

TABLE table-name [table-attributes] type-description ;

The following table-declaration is an example:

TABLE SCORE (1000) U 5;

The table SCORE contains 1001 unnamed entries. These entries may be referenced as SCORE(0), SCORE(1), and so on. The type of these references is table and so their use is limited.

SECTION 2

DIMENSIONED TABLES

SOFTECH

DIMENSIONED TABLES

A JOVIAL (J73) table may have up to seven dimensions. The dimension list appears as part of the table-attributes. The form is

(dimension, ...)

A dimension may have a lower and upper-bound or just an upper-bound. If the lower bound is not specified it defaults to zero (where the upper-bound is an integer S or U) or to the first status-constant of a default status list (where the upper-bound is a status-constant). The lower-bound must be less than or equal to the upper-bound. Lower and upper-bounds must be equivalent or implicitly convertible to each other.

Consider the following examples:

```
TABLE DIMENSIONS (1:5);
  BEGIN
  ITEM LENGTH U;
  ITEM WIDTH U;
  ITEM HEIGHT U;
  END
```

DIMENSIONS

LENGTH(1)	WIDTH(1)	HEIGHT(1)	First Entry
LENGTH(2)	WIDTH(2)	HEIGHT(2)	Second Entry
LENGTH(3)	WIDTH(3)	HEIGHT(3)	Third Entry
LENGTH(4)	WIDTH(4)	HEIGHT(4)	Fourth Entry
LENGTH(5)	WIDTH(5)	HEIGHT(5)	Fifth Entry

Table DIMENSIONS has one dimension which ranges from 1 to 5. These are five entries in DIMENSIONS. Each entry contains a LENGTH, WIDTH and HEIGHT.

```
TABLE DIMENSIONS (2 : 5, 1);
    BEGIN
    ITEM LENGTH U;
    ITEM WIDTH U;
    ITEM HEIGHT U;
    END
```

DIMENSIONS

LENGTH (2,0)	WIDTH (2,0)	HEIGHT (2,0)	ENTRY (2,0)
LENGTH (2,1)	WIDTH (2,1)	HEIGHT (2,1)	ENTRY (2,1)
LENGTH (3,0)	WIDTH (3,0)	HEIGHT (3,0)	ENTRY (3,0)
LENGTH (3,1)	WIDTH (3,1)	HEIGHT (3,1)	ENTRY (3,1)
LENGTH (4,0)	WIDTH (4,0)	HEIGHT (4,0)	ENTRY (4,0)
LENGTH (4,1)	WIDTH (4,1)	HEIGHT (4,1)	ENTRY (4,1)
LENGTH (5,0)	WIDTH (5,0)	HEIGHT (5,0)	ENTRY (5,0)
LENGTH (5,1)	WIDTH (5,1)	HEIGHT (5,1)	ENTRY (5,1)

The table DIMENSIONS now has two dimensions, eight entries, three items in each entry.

```
TYPE GRADE STATUS (V(A),
  V(B), V(C), V(D), V(F));
TABLE CONDITION (V(A) : V(F),
1 : 5);
       BEGIN
       ITEM NAME C 20;
       ITEM AGE U;
       END
```

CONDITION

NAME (V(A), 1)	AGE (V(A), 1)
NAME (V(A), 2)	AGE (V(A), 2)
NAME (V(A), 3)	AGE (V(A), 3)
NAME (V(A), 4)	AGE (V(A), 4)
NAME (V(A), 5)	AGE (V(A), 5)
NAME (V(B), 1)	AGE (V(B), 1)
⋮	
NAME (V(B), 5)	AGE (V(B), 5)
NAME (V(C), 1)	AGE (V(C), 1)
⋮	
NAME (V(F), 5)	AGE (V(F), 5)

Table CONDITION has two dimensions. The first ranges from V(A) to V(F), the second from 1 to 5. There are twenty-five entries in CONDITION, each of which constains a NAME and AGE. (Remember that when a default STATUS type is declared, each status constant in the symbol table is associated with an integer value. This is done positionally from zero.)

DIMENSIONS -- EXERCISES

The following table-declarations are incorrect. Make the necessary corrections.

Incorrect Table-Declarations	Corrections

1. ```
   TABLE HITS (-5 : -10);
      ITEM VALUE U;
   ```

2. ```
   TABLE SCORES (3 : 5,
   1.2 : 4);
      ITEM VALUE U;
   ```

3. ```
   TYPE LAMP STATUS (V(OFF),
   V(WEAK), V(ON)): TABLE
   HOUSES (V(OFF) : 7, -7, :
   0, 3);
      ITEM VALUE U;
   ```

4. ```
   TABLE WORK (1 : 4, 1 : 5, 1
   : 4, 1 : 3, 2, 4, 2, 1 : 2);
      ITEM VALUE U ;
   ```

5. ```
   TABLE NAMES (V(END), 7,
   -2);
      ITEM VALUE U;
   ```

6. ```
   TABLE MALES (7.0 : 10.0);
      ITEM VALUE U ;
   ```

7. ```
   TYPE COLOR STATUS
   (3V(RED), 6V(BLUE),
      4V(YELLOW));
   TABLE HUES (V(RED) :
   V(YELLOW));
   ITEM VALUE U;
   ```
   ```

ANSWERS

The following table-declarations are incorrect. Make the necessary corrections.

Incorrect Table-Declarations	Corrections
1. TABLE HITS (-5 : -10); ITEM VALUE U;	(upper bound must be > lower bound)
2. TABLE SCORES (3 : 5, 1.2 : 4); ITEM VALUE U;	(bounds must be integer or default status)
3. TYPE LAMP STATUS (V(OFF), V(WEAK), V(ON)): TABLE HOUSES (V(OFF) : 7, -7, : 0, 3); ITEM VALUE U;	(lower and upper bound must be same type) (V(OFF) : (V(ON))
4. TABLE WORK (1 : 4, 1 : 5, 1 : 4, 1 : 3, 2, 4, 2, 1 : 2); ITEM VALUE U ;	(only seven dimensions allowed)
5. TABLE NAMES (V(END), 7, -2); ITEM VALUE U;	(default lower bound for integers is zero, 0 : -2 is illegal)
6. TABLE MALES (7.0 : 10.0); ITEM VALUE U;	(bounds must be integer or default status)
7. TYPE COLOR STATUS (3V(RED), 6V(BLUE), 4V(YELLOW)); TABLE HUES (V(RED) : V(YELLOW)); ITEM VALUE U;	(only default status allowed as dimensions)

SUBSCRIPTS

A subscript reference is used to reference an item in a table. The form is:

```
name (subscript-list)
```

A subscript-list has the form

```
(subscript, ... )
```

The subscript list must match the dimension-list in the number of subscripts and the type of subscripts. The subscripts may be values or formulae of the appropriate type and must be within the range of the dimensions.

A subscripted data reference accesses a single item in a dimensioned table. The subscripted item may be used anywhere a simple item may be used.

Given the following declaration:

```
TABLE DIMENSIONS (2 : 5, 1);
      BEGIN
      ITEM LENGTH U;
      ITEM WIDTH U;
      ITEM HEIGHT U;
      END
```

The following statements are correct uses of the subscripted items:

```
COUNT = LENGTH (I, J);

ANSWER = LENGTH (2, 1) * 4 + 7;

FLOAT'VAL = (* F 20 *) (HEIGHT (INDEX, INDEX-1));

TOTAL = -WIDTH (AA + 2, AA);

FOR INDEX :  HEIGHT (-ALPHA, GAMMA**2) BY LENGTH
      (2, 1) WHILE INDEX <= 10;
```

SUBSCRIPTS -- EXERCISES

Given the following declarations:

```
TYPE LIGHT STATUS (V(RED), V(YELLOW), V(GREEN);
TABLE SIGNALS (V(GREEN), 1 : 5);
    BEGIN
    ITEM SYSTEM S;
    ITEM CONTROLS S;
    END
```

The following data references are incorrect. Make the necessary corrections.

Incorrect Reference	Corrections
1. SYSTEM (V(GREEN)	
2. SYSTEM (V(YELLOW), 3.0)	
3. SYSTEM (V(RED), 8)	
4. CONTROLS (V(RED), (V(GREEN));	
5. CONTROLS (2, 3, 5);	

ANSWERS

Given the following declarations:

```
TYPE LIGHT STATUS (V(RED), V(YELLOW), V(GREEN);
TABLE SIGNALS (V(GREEN), 1 : 5);
        BEGIN
        ITEM SYSTEM S;
        ITEM CONTROLS S;
        END
```

The following data references are incorrect. Make the necessary corrections.

	Incorrect Reference	Corrections
1.	SYSTEM (V(GREEN)	SYSTEM (V(GREEN), 4)
2.	SYSTEM (V(YELLOW), 3.0)	SYSTEM (V(YELLOW), 3)
3.	SYSTEM (V(RED), 8)	SYSTEM (V(RED), 5)
4.	CONTROLS (V(RED), (V(GREEN));	CONTROLS (V(RED), 2)
5.	CONTROLS (2, 3, 5);	CONTROLS (V(YELLOW), 5)

SUBSCRIPTS -- EXERCISES

Given the following declarations:

```
TAPE LETTER STATUS (V(A), V(B), V(C), V(D));
TABLE DIMENSIONS (7, 4, 2);
       BEGIN
       ITEM LENGTH F;
       ITEM WIDTH F;
       END
TABLE VALUE (1 : 3, V(B) ; V(D));
       BEGIN
       ITEM COST U;
       ITEM PRICE U;
       ITEM SAVINGS S;
       END
```

The following data references are incorrect. Make the necessary corrections.

Incorrect Data Reference		Corrections
1.	LENGTH (-4, 2, 1)	
2.	WIDTH (3, 4)	
3.	COST (2.0, V(B))	
4.	PRICE (3, V(A))	
5.	SAVINGS (0, V(C), 2)	
6.	COST (3, 3)	

ANSWERS

Given the following declarations:

```
TAPE LETTER STATUS (V(A), V(B), V(C), V(D));
TABLE DIMENSIONS (7, 4, 2);
        BEGIN
        ITEM LENGTH F;
        ITEM WIDTH F;
        END
TABLE VALUE (1 : 3, V(B) ; V(D));
        BEGIN
        ITEM COST U;
        ITEM PRICE U;
        ITEM SAVINGS S;
        END
```

The following data references are incorrect. Make the necessary corrections.

Incorrect Data Reference		Corrections
1.	LENGTH (-4, 2, 1)	LENGTH (4, 2, 1)
2.	WIDTH (3, 4)	WIDTH (3, 4, 1)
3.	COST (2.0, V(B))	COST (2, V(B))
4.	PRICE (3, V(A))	PRICE (3, V(C))
5.	SAVINGS (0, V(C), 2)	SAVINGS (1, V(C))
6.	COST (3, 3)	COST (3, V(D))

PRESETS

As with simple tables, a dimensioned table may contain preset entries. The preset may be given as part of the entry-description or as part of the table attributes. For example, the following declarations result in the table diagrammed below:

```
TYPE LAMP STATUS (V(ON), V(OFF)):

TABLE DIMENSIONS (71 :72, V(ON) : V(OFF)) = 12(o);

        BEGIN

        ITEM LENGTH U;

        ITEM WIDTH U;

        ITEM HEIGHT U;

        END

TABLE DIMENSIONS (71 : 72, V(ON) : V(OFF));

        BEGIN

        ITEM LENGTH U = 4(0);

        ITEM WIDTH U = 4(0);

        ITEM HEIGHT U = 4(0);

        END
```

DIMENSIONS

LENGTH (71, V(ON)) = 0	WIDTH (71, V(ON)) = 0	HEIGHT (71, V(ON)) =0
LENGTH (71, V(OFF)) = 0	WIDTH (71, V(OFF)) = 0	HEIGHT (71, V(OFF)) = 0
LENGTH (72, V(ON)) = 0	WIDTH (72, V(ON)) = 0	HEIGHT (72, V(ON)) = 0
LENGTH (72, V(OFF)) = 0	WIDTH (72, V(OFF)) = 0	HEIGHT (72, V(OFF)) = 0

> *NOTE: The type of the preset must be equivalent or implicitly convertible to the type of the item being preset.*

An omitted value in a table-preset is not initialized. That item has unknown value.

Example

```
        TABLE STUDENTS (1: 10);

                ITEM GRADE C = 2('A'),,
                2('B',), 'C';
```

OR

```
        TABLE STUDENTS (1 :10) =
        2('A'),,2('B',), 'C'

                ITEM GRADE C;
```

STUDENTS
GRADE(1) = 'A'
GRADE(2) = 'A'
GRADE(3) = ?
GRADE(4) = 'B'
GRADE(5) = ?
GRADE(6) = 'B'
GRADE(7) = ?
GRADE(8) = 'C'
GRADE(9) = ?
GRADE(10) = ?

TABLE-PRESET -- EXERCISES

Make a diagram of the table declared below and fill in the values supplied by the table-preset:

```
TABLE SQUARE (3, 3) = POS(0, 2) : -3, 4, 6,, -2, POS(1,2)
  : , 2(, -7), (POS(2, 2) : 3 (-4,, 2(1));

      BEGIN

      ITEM EVENS S;

      ITEM ODDS S;

      END
```

ANSWERS

Make a diagram of the table declared below and fill in the values supplied by the table-preset:

```
TABLE SQUARE (3, 3) = POS(0, 2) : -3, 4, 6,, -2, POS(1,2)
  : , 2(, -7), (POS(2, 2) : 3 (-4,, 2(1));

      BEGIN

      ITEM EVENS S;

      ITEM ODDS S;

      END
```

(0,0)	EV	-	OD	-
(0,1)	EV	-	OD	-
(0,2)	EV	-3	OD	4
(0,3)	EV	6	OD	-
(1,0)	EV	-2	OD	-
(1,1)	EV	-	OD	-
(1,2)	EV	-	OD	-
(1,3)	EV	-7	OD	-

EV	-7	OD	-	(2,0)
EV	-	OD	-	(2,1)
EV	-4	OD	-	(2,2)
EV	1	OD	1	(2,3)
EV	-4	OD	-	(3,0)
EV	1	OD	1	(3,1)
EV	-4	OD	-	(3,2)
EV	1	OD	1	(3,3)

If the preset is a part of the table-attributes, all items within the first entry are preset, then all items within the second entry, ... etc.

Example

```
TABLE ROSTER (1 : 5, 1 : 5) = 10 (, , 16);
      BEGIN
      ITEM NAME C 20;
      ITEM S'S'N C 9;
      ITEM AGE U 7;
      END
```

In the first entry of ROSTER, NAME (1, 1) is not preset, S'S'N (1, 1) is not preset, AGE (1, 1) is preset to 16; in the second entry of ROSTER, NAME (1, 2) is not preset, S'S'N (1, 2) is not preset, AGE (1, 2) is preset to 16; ... in the tenth entry of ROSTER, NAME (2, 5) is not preset, S'S'N (2, 5) is not preset, AGE (2, 5) is preset to 16. All remaining items are not preset.

This may have been written in the following way:

```
TABLE ROSTER (1 : 5, 1 : 5);
      BEGIN
      ITEM NAME C 20;
      ITEM S'S'N C 9;
      ITEM AGE U 7 = 10 (16);
      END
```

A positioner may be used in a dimensioned table-preset to select an entry to begin the preset. The form is:

POS (index-list) : value, ...

The form of the index-list is:

index, ...

> *NOTE: The index-list must match the dimension-list in number of indices and type of indices. The indices may be values or formulas of the appropriate type known at compile-time and must be within the range of the dimensions.*

Example

```
TYPE ALPHA STATUS (V(A), V(B), V(C), V(D), V(E), V(F);
  V(G));

TABLE TWO'D (V(B) : V(F)) = POS (V(C)) : 2(10), 2(10),
  POS(V(F)) : 10;

       BEGIN

       ITEM LENGTH U;

       ITEM WIDTH U;

       END
```

Table TWO'D could be diagrammed as follows:

TWO'D

LENGTH(V(B))	**WIDTH(V(B))**
LENGTH(V(C)) = 10	**WIDTH(V(C)) = 10**
LENGTH(V(D)) = 10	**WIDTH(V(D)) = 10**
LENGTH(V(E))	**WIDTH(V(E))**
LENGTH(V(F)) = 10	**WIDTH(V(F))**

CONSTANT TABLES

A dimensioned table may be declared to be constant, in which case the value of the items contained in the table may not be altered during program execution. The form is:

```
CONSTANT  TABLE  name  [table-attributes];

        entry-description
```

The constant table preset may be given either as part of the table-attributes or the entry description. All items within a constant table do not need to be preset, though these items will have unknown value during execution.

Example

```
CONSTANT TABLE LOOKUP (1 : 10);

        BEGIN

        ITEM NUMBER F = 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0,
          8.0, 9.0, 10.0;

        ITEM SQR F = 1.0, 4.0,, 16.0, POS(8) : 64.0;

        ITEM CUBE F = 1.0, 8.0,, 64.0, POS(8) : 512.0;

        ITEM RECIP F = 1.0, .5,, .25, POS(8) : .125;

        END
```

TABLE TYPE-DECLARATIONS

As discussed in Section 1, a table type-declaration declares a table-type-name which may be used in a table-declaration. The form is:

```
TYPE  table-type-name  TABLE  [dimension-list];
        entry description
```

Once declared, a table-type-name may be used as follows:

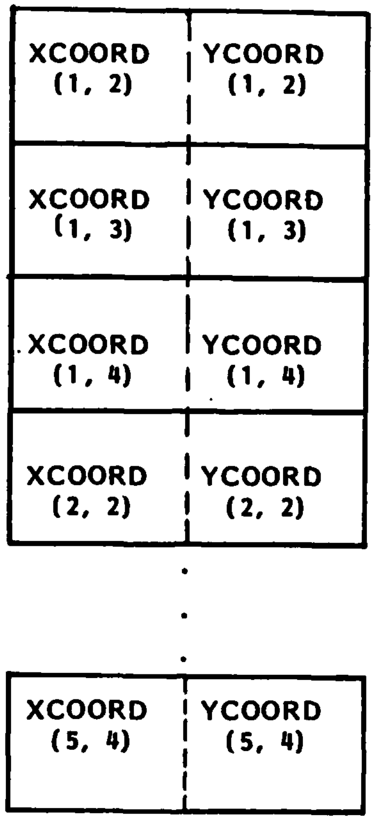
```
TABLE  name  [(dimension-list)]  table-type-name;
```

Consider the following type-declaration:

```
TYPE GRID TABLE (1 : 5, 2 : 4);
        BEGIN
        ITEM XCOORD F;
        ITEM YCOORD F;
        END
```

GRID creates a template like the one diagrammed below. Given a table-declaration like TABLE SQUARES GRID, SQUARES will be laid out according to the template of GRID.

GRID



If the programmer wishes to preset a table declared using a table type name, the preset must be part of the table attributes.

A table may have one and only one dimension-list.

The dimension-list may be a part of either the type-declaration or the table-declaration.

Example

```
TYPE ID TABLE;
        BEGIN
        ITEM NAME C 20;
        ITEM AGE U 7;
```

```
        ITEM HEIGHT F 15;
        END
TABLE CLASS (1 : 200) ID;
```

ID

NAME	AGE	HEIGHT

CLASS

NAME(1)	AGE(1)	HEIGHT(1)
⋮	⋮	⋮
NAME(100)	AGE(100)	HEIGHT(100)

In the above example, the table CLASS has one hundred entries, each of those entries is of type ID. Each of those one hundred entries has NAME, AGE, and HEIGHT.

```
TYPE PAYCHECK TABLE;
      BEGIN
      ITEM REG'PAY A 10, 7;
      ITEM OT'PAY A 10, 7;
      END
```

PAYCHECK

REG'PAY	OT'PAY

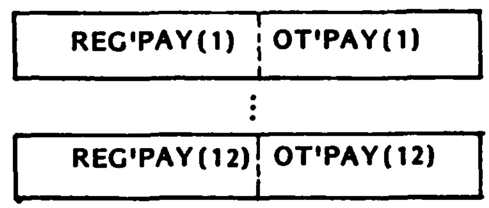
```
TYPE BILLING TABLE (1 : 12 )
  PAYCHECK;
```

BILLING

REG'PAY(1)	OT'PAY(1)
⋮	⋮
REG'PAY(12)	OT'PAY(12)

TABLE OVERHEAD BILLING;

OVERHEAD



> *NOTE: Pointers must be used to reference items in tables declared using a table-type-name. Pointers will be discussed in Workbook 5.*

LIKE-OPTION

Dimensioned table type-declarations may also include a like-option. A like-option allows the programmer to create several variable length records from a common beginning. The form is:

```
TYPE table-type-name TABLE LIKE table-type-name;
     entry-description
```

> *NOTE: An entry-description or a null statement (;) is required.*

A table type-declaration may have one and only one dimension-list. If the like-option refers to a dimensioned table type, the type-declaration itself may not have a dimension list. If the type-description has a dimension-list, a table being declared with that table-type-name must not also have a dimension-list.

For example, consider the following type declarations as diagrammed in Figure 2-2.

```
TYPE BIBLIOGRAPHY TABLE(99);
    BEGIN
    ITEM BOOKNO U;
    ITEM TITLE C 20;
    END
TYPE LIBRARY TABLE LIKE BIBLIOGRAPHY;
    BEGIN
    ITEM COPIES U;
    ITEM LOCATION C 4;
    END
TYPE CATALOG TABLE LIKE LIBRARY;
    BEGIN
    ITEM PUBLISHER C 12;
    ITEM ILLUSTRATED B;
    ITEM AUTHOR C 20;
    END
```

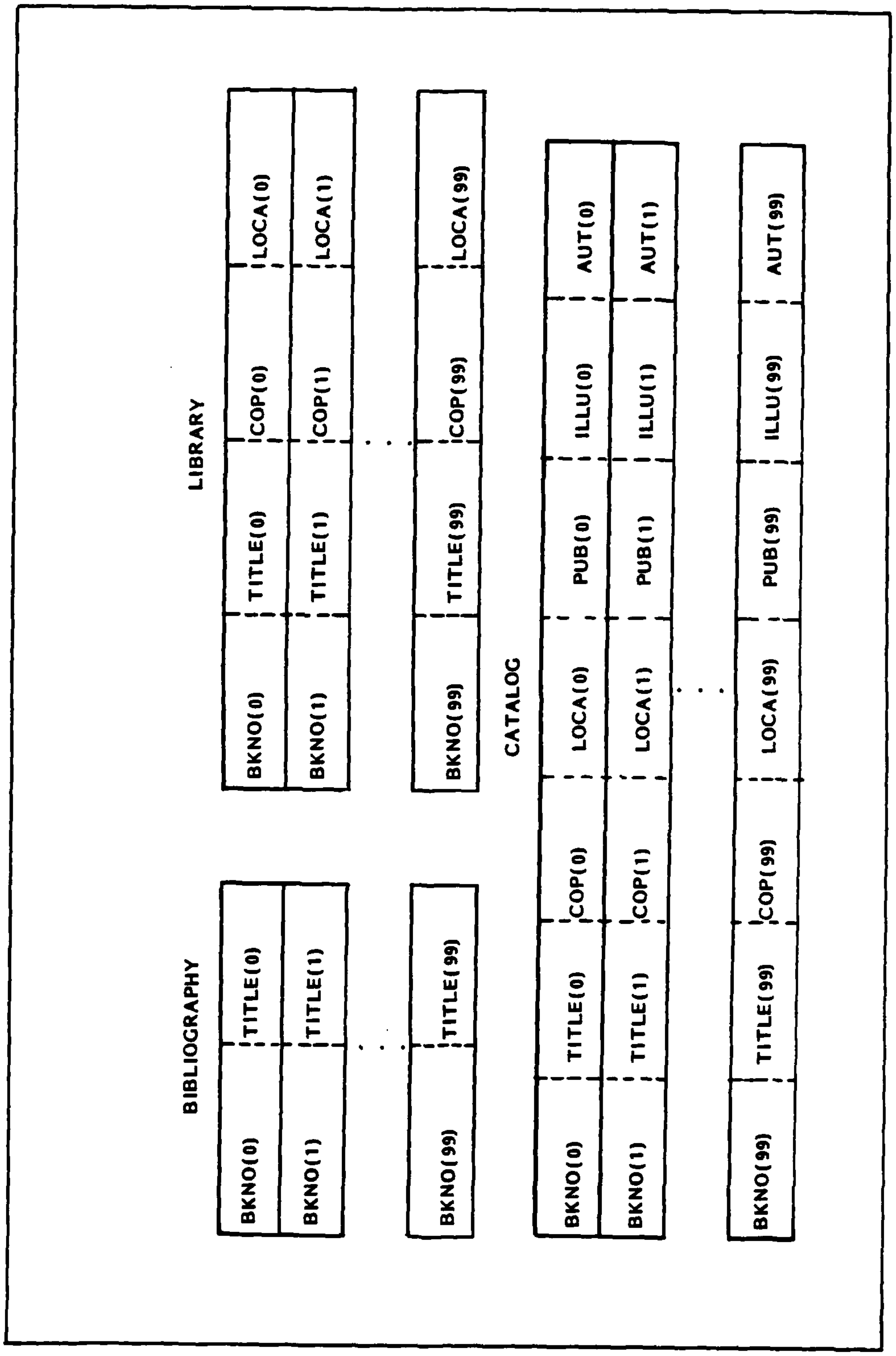


Figure 2-2. Like Option Table Layout

TABLE TYPE-DECLARATIONS -- EXERCISES

Given the following declarations:

```
TYPE SHORTSTRING C 5;
TYPE MEDSTRING C 8;
TYPE LONGSTRING C 12;
TYPE IDENTIFICATION TABLE;
    BEGIN
    ITEM NAME LONGSTRING;
    ITEM RANK SHORTSTRING;
    ITEM SERIALNO LONGSTRING;
    ITEM ENLISTDATE MEDSTRING;
    ITEM ACTIVE'FLAT B;
    END
TYPE INACTIVE TABLE
    LIKE IDENTIFICATION;
    ITEM RETIREDATE MEDSTRING;
TYPE ACTIVE TABLE
    LIKE IDENTIFICATION;
    BEGIN
    ITEM SERVICEPERIOD U;
    ITEM ASSIGNMENT U;
    END
```

For the following table-declarations, draw a diagram showing the items in each entry and any preset values.

1. TABLE SMITH IDENTIFICATION;

2. TABLE JONES ACTIVE;

3. TABLE PRESCOTT IDENTIFICATION = 'PRESCOTT',,'123',,TRUE;

4. TABLE COMPANY'C (20) INACTIVE;

5. Declare a table that includes marital and education status for active personnel.

ANSWERS

1. SMITH

NAME	RANK	SERIAL'NO	ENLISTDATE	ACTIVE'FLAG
C 12	C 5	C 12	C 8	B

2. JONES

NAME	RANK	SERIAL'NO	ENLISTDATE	ACTIVE'FLAG	SERVICE PER	ASSIGNMENT
C 12	C 5	C 12	C 8	B	U	U

3. PRESCOTT

NAME	RANK	SERIAL'NO	ENLISTDATE	ACTIVE'FLAG
PRESCOTT	–	123	–	TRUE

4. COMPANY'C

NAME	RANK(0)	SER'NO(0)	ENLDATE(0)	ACT'FL(0)	RETIRE(0)

⋮

NAME(20)	RANK(20)	SER'NO(20)	ENLDATE(20)	ACT'FL(20)	RETIRE(20)

5. TYPE COMPLETE'ID TABLE LIKE ACTIVE;

```
BEGIN
ITEM MARRIED B;
ITEM EDUC STATUS (V(GS), V(HS), V(TECH, V(BACH), V(GRAD));
END
```

TABLE JOHN'DOE COMPLETE'ID;

NAME	RANK	SER'NO	ENLDATE	ACT'FL	SERVP	ASSIG	MARRIED	EDUC

TABLE TYPE EQUIVALENCE AND IMPLICIT CONVERSION

The following is a complete list of requirements for table type equivalence and implicit conversion for simple and dimensioned tables. Certain of these concepts have not been discussed yet, but are included here for completeness and ease of reference.

Two tables have equivalent types if:

- Their structure specifiers are the same
- They have the same number of dimensions
- They have the same number of elements in each dimension
- They have the same number of items in each entry
- The types (including attributes) and the textual order of the items are equivalent
- The explicit or implied packing-spec on each of these items is the same, and
- The !ORDER directive is either present or absent in both tables

The names of the items, as well as the types and bounds of the dimensions, need not be the same for the tables to be equivalent.

A table entry is considered to have no dimensions.

A table whose entry contains an item-declaration is not considered equivalent to a table whose entry is declared using an unnamed item-description.

EXPLICIT CONVERSION

A table may be explicitly converted to a bit string through the use of the REP built-in function. The conversion of a bit string to a table type is legal only if the size of the bit string is equal to the BITSIZE of the REP of the table type.

An explicit conversion operator may be applied to a table to assert its type for the benefit of the reader. The conversion-operator may not change the type of the table.

Example

```
TYPE SQUARE TABLE;
     ITEM VALUE F;
TABLE NUMBERS SQUARE;
... (* SQUARE *)(NUMBERS) ...
```

TABLE DECLARATIONS -- EXERCISES

1. Declare a table of 100 entries, each entry to be made up of variables to be assigned the following data:

   a. A person's name,

   b. A person's age,

   c. A person's height, in centimeters, always exact to the nearest tenth of a centimeter (i.e., a millimeter),

   d. A flag to indicate if married or not,

   e. The state of a person's eye color,

   f. A person's grade point average,

   g. The number of whole grams that a person's weight differs from the average.

2. Declare a constant table with components equal to the square roots of the integers 1 through 10.

3. Given the following declarations:

   ```
   TYPE LETTERS STATUS (V(A), V(B), V(C), V(D),
   V(E), V(F), V(G), V(H), V(I), V(J),

   TABLE RESULTS(9, V(J));
   ITEM DATA POINT U;
   ```

   a. Initialize every other component to 1.

   b. Initialize every fifth component to 2.

   c. Initialize the four corners to 3.

   d. Initialize DATAPOINT(6, V(G)) through DATAPOINT(7, V(C)) to 4.

ANSWERS

```
1.    TABLE ID (1 : 100);
            BEGIN
            ITEM NAME C 20;
            ITEM AGE U;
            ITEM HEIGHT A 9, 4;
            ITEM MARRIED B;
            ITEM EYES STATUS (V(BROWN), V(BLUE));
            ITEM AVE F;
            ITEM DEVIATION S;
            END

2.    CONSTANT TABLE SQUAREROOTS (1 : 10);
            ITEM ROOT F = 1., 2.** .5, 3.** .5, 2., 5.**.5,
                          6.** .5, 7.** .5, 8.** .5, 3., 10.** .5;

3a.   TABLE RESULTS (9, V(J)) = 50(1, );

3b.   TABLE RESULTS (9, V(J)) = 20(, , , , 2);

3c.   TABLE RESULTS (9, V(J)) = POS(0, V(A)) : 3, POS(9, V(A)) : 3,
                                POS(0, V(J)) : 3, POS(9, V(J)) : 3;

3d.   TABLE RESULTS (9, V(J)) = POS(6, V(G)) : 7(4);
```

SECTION 3

BLOCKS

BLOCKS

A block is a collection of values gathered into one region of memory. These values can be items, tables or nested blocks. Blocks are useful in memory management. For instance, certain data that must be paged in and out of memory together may be placed in a block. Because the items, tables and blocks are enclosed in a block, the compiler allocates them together. However, the compiler is free to allocate the data within the block in any order. (If order is important, use the !ORDER directive.)

A block-declaration has the form:

```
BLOCK name ;
      block-body
```

A block-body is:

$$\left\{ \begin{array}{l} \text{Item-declaration} \\ \text{table-declaration} \\ \text{block-declaration} \end{array} \right\} \ \ldots$$

Examples

a.

```
BLOCK INTERFACE;
      BEGIN
      ITEM CHANNEL U;
      ITEM FREQUENCY A 6, 2;
      TABLE PARS (9);
            BEGIN
            ITEM SETTING F;
            ITEM DIAL S 10;
            END
      END
```

b. BLOCK DATAGROUP;

```
BEGIN
ITEM COUNT U;
BLOCK INFO;
    BEGIN
    TABLE LIBRARY (1 : 10);
        BEGIN
        ITEM BOOK C 20;
        ITEM AUTHOR C 20;
        ITEM PRESENT B 1;
        END
    ITEM EMPLOYEES U 20;
    ITEM COST A 10, 7;
    END
TABLE FACILITY (3, 4, 3);
    ITEM PRICE S;
END
```

When a programmer uses the name DATAGROUP, all the information about all the items, tables, and nested blocks becomes available.

NOTE: A block may be "simple" (either one data-declaration or a null declaration) though a simple block-body has no real role in the language.

Examples

```
BLOCK SMALL1;
    ITEM BOOKS U 10;
BLOCK SMALL 2; ;
BLOCK SMALL 3;
    BLOCK DATA;
        TABLE NUMBERS;
            BEGIN
            ITEM AGE U;
            ITEM HEIGHT F;
            END
```

BLOCK TYPE-DECLARATIONS

A block type-declaration declares a block-type-name that may be used in a block-declaration. The form is:

TYPE block-type-name BLOCK

block body

The block type-name may then be used in a block-declaration:

BLOCK name block-type-name [block-preset];

Example

```
TYPE GROUP BLOCK
      BEGIN
      ITEM COUNT S;
      TABLE POSITION (1 : 20);
            BEGIN
            ITEM XCOORD F;
            ITEM YCOORD F;
            END
      BLOCK REGISTER;
            BEGIN
            ITEM NAME C 10;
            TABLE PLACE (1 : 3, 1 : 4);
                  ITEM NUMBER S;
            END
      END
BLOCK MAPPINGS GROUP;
BLOCK XX;
      BEGIN
      ITEM TOTAL S;
      BLOCK INFO'SET GROUP;
      END
```

THE JOVIAL (J73) WORKBOOK

VOLUME 5

POINTERS

1081-1

April 1981

Submitted to

Department of the Air Force
Rome Air Development Center
I S I S
Griffiss Air Force Base, NY 13441

Prepared by

SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02154

PREFACE

Workbook 5 is intended for use with Tape 5 of the JOVIAL (J73) Video Course. Its purpose is to elaborate upon and reinforce concepts and language rules introduced in the videotape.

This workbook discusses pointer item declarations, dereferences, pointer qualified references, the setting of pointer values, and the LOC function. A summary of the material presented can be found in Section 3.

TABLE OF CONTENTS

SYNTAX

The syntax conventions used in the JOVIAL (J73) Video Course and workbook are as follows:

Syntax	Meaning	Sample Expansions
[some-feature]	Brackets indicate an optional feature.	some-feature OR nothing
{one | other}	Braces with a vertical bar indicate disjunction- a choice between alternatives.	one OR other
{this-one that-one}	Braces with each feature on separate lines indicate disjunction - a choice between alternatives.	this-one OR that-one
letter ...	The sequence '...' indicates one or more repetitions of a feature.	letter letter letter letter letter letter ...
(letter),...	The sequence "..." following a comma (or a colon) indicates one or more repetitions of the feature separated by commas (or colons).	(letter) (letter) (letter) (letter) (letter) (letter) ...
[{this-one that-one}] + another	Syntax symbols may be combined.	this-one + (another) that-one + (another) + (another)

SECTION 1

POINTERS

SOFTECH

POINTERS

A pointer is an item whose value is a machine address. They are used to indirectly reference data. Pointers are often thought confusing by the novice programmer though without good reason. With a little practice, the use of pointers will become second nature and provide a powerful means of exploiting the full potential of JOVIAL (J73).

The general syntax for pointer item declarations is of the same form as all other item declarations:

$$[\text{CONSTANT}]\ \text{item}\ \text{name}\ \left\{ \begin{array}{l} \text{type-description} \\ \text{item-type-name} \end{array} \right\} [\text{item-preset}];$$

ITEM — ITEM is a JOVIAL (J73) reserved word. It has a specific meaning to the compiler, which restricts its use to declarations. It cannot be used as a name.

name — A name is a sequence of letters, digits, dollar signs and primes (single quote characters). It must begin with a letter or a dollar sign, and be at least two characters long. A name may be any length, though only the first 31 characters are examined for uniqueness within the program. (Fewer than 31 characters may be examined for external names. External names are discussed in Workbook 8).[1]

type description — A pointer type-description has the form

P [type-name]

A pointer followed by a type-name references items only of that type. That pointer is said to be "typed." A pointer not followed by a type-name references items of any type. That pointer is said to be "untyped." An untyped pointer has limited uses and will not be discussed.

; — A semi-colon terminates the declaration.

---

[1]See list of JOVIAL (J73) reserved words given in Workbook 1, p. 3-2.

POINTER ITEM PRESETS

JOVIAL (J73) has only one pointer literal - NULL. Any pointer, typed or untyped, may be preset to NULL or receive the value NULL in an assignment.

Examples

```
ITEM MAXPTR P MAX = NULL;

ITEM LISTPTR P = NULL;
```

CONSTANT POINTER ITEM-DECLARATION

A pointer item may be declared to be of constant value. In this case the reserved word CONSTANT and an item preset (i.e., NULL) are required.

Examples

```
CONSTANT ITEM MAXPTR P MAX = NULL;

CONSTANT ITEM LISTPTR P = NULL;
```

Pointer Item Declarations: Examples

ITEM MAXPTR P MAX;	Item MAXPTR is a pointer which references data only of type MAX.
ITEM VALPTR P VALUE;	Item VALPTR is a pointer which references data only of type VALUE.
CONSTANT ITEM ENDPTR P = NULL;	Item ENDPTR is a pointer item with a constant value of NULL.
ITEM FOOPTR P;	Item FOOPTR is an untyped pointer. It can contain the address of a data object of any type.

POINTER DEREFERENCE

A pointer dereference is of the form:

@ pointer-name
(pointer-formula)

A pointer dereference results in a data object of the type pointed to by the pointer.

Examples

1) TYPE MAXVAL S;

ITEM MAXPTR P MAXVAL;

Dereference	Meaning
@ MAXPTR	This pointer dereference results in an item of type MAXVAL.

2) TYPE MAXVAL S;

ITEM MAXPTR P MAXVAL;

@ MAXPTR = 2;	The value 2 is assigned to the item pointed to by MAXPTR.
@ MAXPTR = @ MAXPTR + 1;	The item pointed to by MAXPTR is incremented by one and assigned back into the item pointed to by MAXPTR.

3) TYPE SQUARE TABLE;

```
BEGIN
ITEM LENGTH U;
ITEM WIDTH U;
END
```

ITEM SQPTR P SQUARE;

Dereference	Meaning
@ SQPTR	This pointer dereference "results" in a table of type SQUARE - a table with one entry with two items, LENGTH, WIDTH. (Of course, SQPTR does not point to a physical data object until a table has been declared using square as a type description.)

4) TYPE STATS BLOCK

```
TYPE STATS BLOCK
    BEGIN
    TABLE PRICE;
        BEGIN
        ITEM COST U;
        ITEM VALUE U;
        END
    ITEM PROFIT S;
    END
ITEM STPTR P STATS;
```

Dereference	Meaning
@STPTR	This pointer dereference results in a block of the type STATS - a block with a table, PRICE, and an item, PROFIT. (Again, STPTR does not point to a physical object in storage until a block has been declared using STATS as a type description.)

POINTER QUALIFIED REFERENCES

Any items within a table declared using a table type name or items within blocks declared using a block type name must be accessed by using a pointer. Consider the following situation:

```
TYPE GRID TABLE (1 : 10);
    BEGIN
    ITEM XCOORD S;
    ITEM YCOORD S;
    END
TABLE MAP GRID;
TABLE CHECKERS GRID;
```

The type declaration above creates a table template named GRID. GRID has ten entries, each of which contains items XCOORD and YCOORD. MAP and CHECKERS are declared to be tables that look like GRID. Each table contains ten entries; each entry contains items XCOORD and YCOORD. What happens when a reference is made to XCOORD (3) in a program with both of these tables? Obviously, the reference is ambiguous.

To avoid this problem, a pointer-qualified reference is used to access items in typed tables or blocks. The forms of a pointer-qualified reference are:

name [(subscript-list)] dereference

or

dereference [(subscript-list)]

A dereference is of the form

$$@\left\{\begin{array}{l}\text{POINTER-name}\\ \text{(pointer-formula)}\end{array}\right\}$$

Examples

1)
```
TYPE DATA TABLE;
        BEGIN
        ITEM VALUE U;
        ITEM FIND B;
        END
ITEM DATAPTR P DATA;
```

TYPE DATA

VALUE	FIND

References	Meaning
@ DATAPTR	DATAPTR points to objects of type DATA. The dereference of DATAPTR results in a table with one entry containing two items, VALUE and FIND.
VALUE @ DATAPTR	This data reference results in one item, VALUE from a table of type DATA.

2) TYPE DIMENSIONS TABLE (1 : 15);

```
TYPE DIMENSIONS TABLE (1 : 15);
     BEGIN
     ITEM LENGTH F;
     ITEM WIDTH F;
     ITEM HEIGHT F;
     END
ITEM DIMPTR P DIMENSIONS;
```

TYPE DIMENSIONS

LENGTH(1)	WIDTH(1)	HEIGHT(1)
LENGTH(2)	WIDTH(2)	HEIGHT(2)
⋮	⋮	⋮
LENGTH(15)	WIDTH(15)	HEIGHT(15)

<u>References</u>	<u>Meaning</u>
@ DIMPTR	DIMPTR points to objects of type DIMENSIONS. The dereference of DIMPTR is a <u>table</u> with fifteen entries, each with three items, LENGTH, WIDTH, HEIGHT.
@ DIMPTR(13)	This reference is the entire thirteenth entry, with three items, LENGTH, WIDTH, and HEIGHT.
LENGTH(11) @ DIMPTR	The dereference of DIMPTR results in a table of type DIMENSIONS. This reference is LENGTH(11) from a table of type DIMENSIONS.

3)

```
TYPE COORD TABLE;
     BEGIN
     ITEM LONG S;
     ITEM LAT S;
     END
TYPE POSITION TABLE
(1 : 25) COORD;
ITEM CPTR P COORD;
ITEM PPTR P POSITION;
```

TYPE COORD

LONG	LAT

TYPE POSITION

LONG(1)	LAT(1)
LONG(2)	LAT(2)
⋮	⋮
LONG(25)	LAT(25)

Dereference	Meaning
@ CPTR	CPTR points to objects of type COORD. The dereference of CPTR results in a table with one entry with two items, LONG and LAT.
@ PPTR	PPTR points to objects of type POSITION. The dereference of PPTR results in a table with twenty-five entries each with two items, LONG and LAT.
LAT @ CPTR	This data reference results in one item, LAT, from a table of type COORD.
LONG(20) @ PPTR	This data reference results in one item, LONG(20), from a table of type POSITION.
@ PPTR(4)	This reference results in the entire fourth entry from a table of type POSITION. The entry is composed of LONG and LAT.

One possible use may be:

```
LAT @ CPTR = LONG(20) @ PPTR;
```

POINTERS -- EXERCISES

Given the following declarations:

```
TYPE SHORTSTRING C 6;
TYPE CITATION TABLE;
        BEGIN
        ITEM AUTHOR C 20;
        ITEM DATE SHORTSTRING;
        END
TYPE LIBRARY TABLE (1 : 50);
        BEGIN
        ITEM BOOK'NO SHORTSTRING;
        ITEM COPIES U;
        END


TYPE PROSE TABLE (9) CITATION;
ITEM SPTR P SHORTSTRING;
ITEM CPTR P CITATION;
ITEM LPTR P LIBRARY;
ITEM PPTR P PROSE;
```

TYPE CITATION

AUTHOR	DATE

TYPE LIBRARY

BOOK'NO(1)	COPIES(1)
⋮	⋮
BOOK'NO(50)	COPIES(50)

TYPE PROSE

AUTHOR(0)	DATE(0)
⋮	⋮
AUTHOR(9)	DATE(9)

Determine if the following are correct or incorrect data references.

Reference	C	I
1. 1 CPTR		
2. BOOK'NO(42) @ LPTR		
3. AUTHOR @ CPTR		
4. DATE(4) @ PPTR		
5. @ SPTR		
6. AUTHOR @ PPTR		
7. @ PPTR(9)		
8. DATE(3) @ CPTR		
9. @ LPTR(18)		
10. DATE(14) @ PPTR		
11. COPIES @ LPTR		
12. @ PPTR		
13. COPIES @ LPTR(29)		

ANSWERS

Determine if the following are correct or incorrect data references.

References	C	I
1. @ CPTR	X	
2. BOOK'NO(42) @ LPTR	X	
3. AUTHOR @ CPTR	X	
4. DATE(4) @ PPTR	X	
5. @ SPTR	X	
6. AUTHOR @ PPTR		X (AUTHOR (8) @ PPTR)
7. @ PPTR(9)	X	
8. DATE(3) @ CPTR		X (DATE @ CPTR)
9. @ LPTR(18)	X	
10. DATE(14) @ PPTR		X (subscript out of range)
11. COPIES @ LPTR		X (COPIES(20) @ LPTR)
12. @ PPTR	X	
13. COPIES @ LPTR(29)		X (COPIES(29) @ LPTR)

TYPE EQUIVALENCE, IMPLICIT CONVERSION

Two pointer types are equivalent if they are both untyped pointers or if they are both typed pointers with the identical type attribute.

A typed pointer will be implicitly converted to an untyped pointer.

Declarations

```
TYPE MAX U 7;

ITEM MAX1PTR P MAX;

ITEM ANY1PTR P;

ITEM MAX2PTR P MAX;

ITEM ANY2PTR P;
```

MAX1PTR and MAX2PTR have equivalent types - they are both typed pointers, pointing to items of type MAX.

ANY1PTR and ANY2PTR have equivalent types - they both are untyped pointers.

The type of MAX1PTR may be *IMPLICITLY CONVERTED to the* type of ANY1PTR, not vice-versa.

EXPLICIT CONVERSION

The forms of an explicit conversion are:

$$\left\{\begin{array}{l}\text{(* type-description *)}\\ \text{type-indicator}\\ \text{item-type-name}\\ \text{(* item-type-name *)}\end{array}\right\}\quad\text{(formula)}$$

Pointer types may be explicitly converted to:

P - Converting a pointer to a different pointer type means that the pointer will be considered as a pointer of the specified type.

Example

```
TYPE MAX U 7;

ITEM ANY1PTR P;

ITEM NEWPTR P NEW;

(* P MAX *)(ANY1PTR)

(* P MAX *)(NEWPTR)
```

Data items that may be converted to a pointer type are:

B - Conversion will take place only if the size of the bit string is equal to BITSINPOINTER.

S, U - Converting an integer to a pointer is equivalent to first converting the integer to type B BITSINPOINTER, then converting the bit string to a pointer.

Pointer types may be explicitly converted to:

B - When a pointer type is explicitly converted to type B n, the result is the rightmost n bits of the pointer.

S, U - Converting a pointer to an integer type is equivalent to first converting the pointer to type B BITSINPOINTER, then converting the bit string to an integer.

SECTION 2

THE LOC FUNCTION

SOFTECH

LOC

The LOC function returns the machine address of its argument. The form is

LOC (argument)

Argument may be a data-name, a statement-name or a subroutine-name. The LOC function returns a typed pointer if the argument is declared with a type-name. Otherwise the pointer returned will be untyped.

The LOC function is a pointer formula. The value it returns may be assigned to a pointer item or may be used in a pointer dereference. (A pointer used in a dereference must be a typed pointer.)

Examples

1)
```
TYPE SQUARE TABLE;
        BEGIN
        ITEM LENGTH U;
        ITEM WIDTH U;
        END
ITEM SQPTR P SQUARE;
TABLE ROOM SQUARE;
```

Expression	Result
LOC(ROOM)	The function returns a pointer of type SQUARE that points to the address of ROOM.
SQPTR = LOC(ROOM);	SQPTR points to the address of ROOM.

2) TYPE COORD TABLE;

```
TYPE COORD TABLE;
        BEGIN
        ITEM LONG S;
        ITEM LAT S;
        END
TABLE SPOT COORD;
TABLE HOUSE (1 : 10, 1 : 10) COORD;
ITEM CITPTR P COORD;
```

Expression	Meaning
CITPTR = LOC(HOUSE(4, 5));	CITPTR points to the address of the entire 4, 5th entry of HOUSE. (Each entry in HOUSE is of type COORD).
CITPTR = LOC(SPOT);	CITPTR points to the address of SPOT.

3)

```
TYPE COORD TABLE;
        BEGIN
        ITEM LONG S;
        ITEM LAT S;
        END
TYPE POSITION TABLE (1 : 25) COORD;
TABLE SPOT COORD;
TABLE WORLD POSITION;
```

Data Reference	Meaning
LAT @ (LOC(SPOT))	LOC of SPOT returns a pointer of type COORD that points to the address of SPOT. The reference is to the item LAT in that table.
@ (LOC(WORLD))	LOC of WORLD returns a pointer of type POSITION that points to the address of WORLD. This reference is to the entire table.
LONG @ (LOC(WORLD(13)))	LOC of WORLD(13) returns a pointer of type COORD that points to the address of the thirteenth entry in WORLD. The reference is to the item LONG in that entry.

THE LOC FUNCTION -- EXERCISES

Given the following declarations:

```
TYPE CITATION TABLE;
     BEGIN
     ITEM AUTHOR C 20;
     ITEM DATE C 6;
     END
TYPE LIBRARY TABLE (1 : 50);
     BEGIN
     ITEM BOOK'NO C 6;
     ITEM COPIES U;
     END
TYPE PROSE TABLE(9) CITATION;
TABLE QUOTE CITATION;
TABLE INFORMATION (1 : 3, 1 : 3) CITATION;
TABLE BOOK'STORE LIBRARY;
TABLE CATALOG PROSE;
```

Determine if the following are correct or incorrect references using the LOC function.

Reference	C	I
1. DATE @ (LOC(QUOTE))		
2. AUTHOR @ (LOC(INFORMATION(2, 3)))		
3. COPIES @ (LOC(BOOK'STORE))		

<u>Reference</u>	<u>C</u>	<u>I</u>
4. DATE(3, 1) @ (LOC(INFORMATION))		
5. DATE(9) @ (LOC(CATALOG))		
6. AUTHOR @ (LOC(CATALOG(4)))		
7. AUTHOR(4) @ (LOC(QUOTE))		
8. AUTHOR @ (LOC(CATALOG(0)))		
9. BOOK'NO(32) @ (LOC(BOOK'STORE))		
10. DATE @ (LOC(BOOK'STORE))		
11. COPIES @ (LOC(BOOK'STORE(4)))		
12. BOOK'NO @ (LOC(QUOTE))		

ANSWERS

Reference	C	I
1. DATE @ (LOC(QUOTE))	X	
2. AUTHOR @ (LOC(INFORMATION(2, 3)))	X	
3. COPIES @ (LOC(BOOK'STORE))		X (COPIES(10) @ (LOC...))
4. DATE(3, 1) @ (LOC(INFORMATION))		X (DATE @ (LOC(INFO (3,1))))
5. DATE(9) @ (LOC(CATALOG))	X	
6. AUTHOR @ (LOC(CATALOG(4)))	X	
7. AUTHOR(4) @ (LOC(QUOTE))		X (AUTHOR @ (LOC (QUOTE)))
8. AUTHOR @ (LOC(CATALOG(0)))	X	
9. BOOK'NO(32) @ (LOC(BOOK'STORE))	X	
10. DATE @ (LOC(BOOK'STORE))		X (DATE not in BOOK'ST)
11. COPIES @ (LOC(BOOK'STORE(4)))		X (COPIES(4) @ (LOC(BK'S)))
12. BOOK'NO @ (LOC(QUOTE))		X (BOOK'NO not in QUOTE)

POINTERS AND THE LOC FUNCTION

By using the LOC function and a pointer, an item in a typed table or a typed block may be correctly referenced.

The first slide in this section presented the problem:

```
TYPE GRID TABLE (1 : 10);
    BEGIN
    ITEM XCOORD S;
    ITEM YCOORD S;
    END
TABLE MAP (1 : 10) GRID;
TABLE CHECKERS GRID;
```

XCOORD(3) is an illegal data reference.

The following declarations and statements clarify the problem:

```
ITEM GPTR P GRID;
GPGR = LOC(MAP);
XCOORD(3) @ GPTR = 7;
```

THE LOC FUNCTION -- EXERCISES

Given the following declarations:

```
TYPE CITATION TABLE;
        BEGIN
        ITEM AUTHOR C 20;
        ITEM DATE C 6;
        END
TYPE LIBRARY TABLE (1 : 50);
        BEGIN
        ITEM BOOK'NO C 6;
        ITEM COPIES U;
        END
TYPE PROSE TABLE(9) CITATION;
TABLE QUOTE CITATION;
TABLE INFORMATION (1 : 3, 1 : 3) CITATION;
TABLE BOOK'STORE LIBRARY;
TABLE CATALOG PROSE;
```

and given the following statements:

```
CPTR = LOC(QUOTE);
LPTR = LOC(BOOK'STORE);
PPTR = LOC(CATALOG);
CITPTR = LOC(CATALOG(8));
IPTR = LOC(INFORMATION(2, 1));
```

Determine if the following data references are correct or incorrect.

Reference	C	I
1. DATE @ CPTR		
2. @ (LOC(BOOK'STORE))(15)		
3. AUTHOR(2, 1) @ IPTR		
4. BOOK'NO(13) @ LPTR		
5. DATE @ IPTR		
6. @ CPTR(7)		
7. COPIES @ LPTR(27)		
8. AUTHOR @ CITPTR		
9. BOOK'NO @ CPTR		
10. DATE @ IPTR		
11. DATE @ (LOC(CATALOG(3)))		
12. COPIES @ (LOC(BOOK'STORE(4)))		

ANSWERS

Reference	C	I
1. DATE @ CPTR	X	
2. @ (LOC(BOOK'STORE))(15)	X	
3. AUTHOR(2, 1) @ IPTR		X (AUTHOR @ IPTR)
4. BOOK'NO(13) @ LPTR	X	
5. DATE @ IPTR	X	
6. @ CPTR(7)		X (@ CPTR)
7. COPIES @ LPTR(27)		X (COPIES(27) @ LPTR)
8. AUTHOR @ CITPTR	X	
9. BOOK'NO @ CPTR		X (BK'NO not part of type CITATION)
10. DATE @ IPTR	X	
11. DATE @ (LOC(CATALOG(3)))	X	
12. COPIES @ (LOC(BOOK'STORE(4)))		X (COPIES(4) @ (LOC(BOOK'STORE)))

POINTER TYPE-DECLARATIONS

A pointer type-declaration creates a template which may then be referred to by name to describe an item as having those attributes. The form is

```
TYPE  item-type-name  type-description;
```

Examples

```
TYPE HEADPTR P;

TYPE BOTTOMPTR P LOWER;

ITEM HPTR HEADPTR;

ITEM TOPPTR HEADPTR = NULL;

CONSTANT ITEM ENDPTR BOTTOMPTR = NULL;

ITEM BPTR BOTTOMPTR;
```

SECTION 3

SUMMARY

POINTERS

A general form of an item-declaration is:

[CONSTANT] ITEM name $\left\{ \begin{array}{l} \text{type-description} \\ \text{item-type-name} \end{array} \right\}$ [item-preset];

A pointer type-description is of the form:

P [type-name]

A pointer, like any other item, may be declared with a previously declared item-type-name.

The value of a pointer is a memory address.

A pointer is either typed or untyped.

A pointer-qualified reference is used to access components of tables or blocks declared using a type-name. The form of a pointer-qualified reference is:

name [(subscript-list)] dereference

Dereference is of the form:

@ $\left\{ \begin{array}{l} \text{pointer-name} \\ \text{(pointer-formula)} \end{array} \right\}$

NOTES: Name must be a component of the type to which the pointer points. The pointer must not be an untyped pointer.

THE LOC FUNCTION

The LOC function returns the machine address of its argument. The form is:

LOC (argument)

The LOC function returns a typed pointer if argument is declared with a type-name.

The LOC function is a pointer expression whose result may be assigned to a pointer variable or used in a dereference.

THE JOVIAL (J73) WORKBOOK

VOLUME 6

SUBROUTINES

1081-1

April 1981

Submitted to

Department of the Air Force
Rome Air Development Center
I S I S
Griffiss Air Force Base, NY 13441

Prepared by

SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02154

PREFACE

This workbook is intended for use with Tape 6 of the JOVIAL (J73) Video Course. Its purpose is to elaborate upon and reinforce concepts and language rules introduced in the videotape.

The definition, invocation and termination of subroutines are discussed in this workbook. Addressed in detail are procedures, functions, parameterization, the use-attribute and parameter binding, with several examples and exercises. The final section is a summary of the information presented herein.

TABLE OF CONTENTS

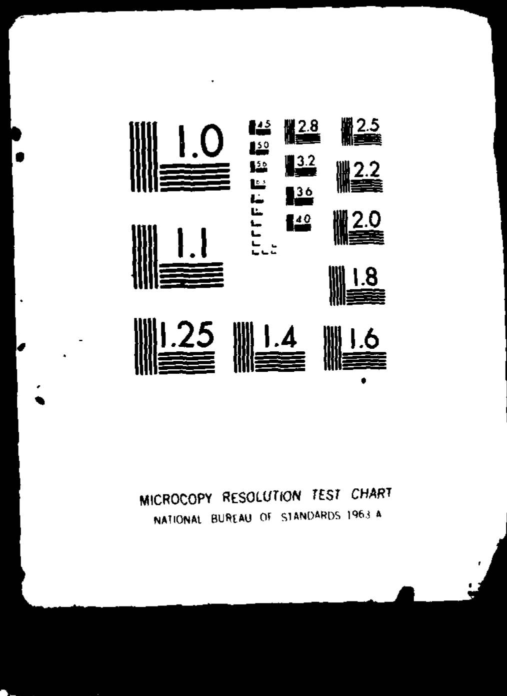
AD-A108 527 SOFTECH INC WALTHAM MA F/G 5/9
THE JOVIAL (J73) WORKBOOK. VOLUME I. INTEGER AND FLOATING POINT--ETC(U)
NOV 81 F30602-79-C-0040
UNCLASSIFIED RADC-TR-81-333-VOL-1 NL

4 of 5

SIFIED

4 OF 5

AD A108527



SYNTAX

The syntax conventions used in the JOVIAL (J73) Video Course and workbook are as follows:

Syntax	Meaning	Sample Expansions
[some-feature]	Brackets indicate an optional feature.	some-feature OR nothing
{one | other}	Braces with a vertical bar indicate disjunction- a choice between alternatives.	one OR other
{this-one that-one}	Braces with each feature on separate lines indicate disjunction - a choice between alternatives.	this-one OR that-one
letter ...	The sequence '...' indicates one or more repetitions of a feature.	letter letter letter letter letter letter ...
(letter),...	The sequence "..." following a comma (or a colon) indicates one or more repetitions of the feature separated by commas (or colons).	(letter) (letter) (letter) (letter) (letter) (letter) ...
[{this-one that-one}] + another	Syntax symbols may be combined.	this-one + (another) that-one + (another) + (another)

SECTION 1

INTRODUCTION

SOFTECH

INTRODUCTION

The form of the main JOVIAL program module is:

```
START PROGRAM name
    BEGIN
    [ declaration ... ]
    executable statement ...
    [ subroutine-definition ... ]
    END
    [ subroutine-definition ... ]
TERM
```

The main program module combines declarations of data, executable statements, and subroutine-definitions in a single file that can be compiled, linked to other modules (if necessary), and executed.

A subroutine is like a small program in the sense that it also contains a declaration section, an executable portion and possibly other subroutines nested within it. They may be used in a program to provide modularity. A subroutine may be used to perform a similar sequence of actions at several different points within a complete program.

Subroutine-definitions can appear in two places, before and after the END. They are optional in both places; if subroutines are needed, then they must be included. The subroutine-definitions before the END are called "nested," and those after END are called "non-nested." Only non-nested subroutines can be designated as external by the use of the reserved word DEF. External subroutines are described in Workbook 8.

```
START
        PROGRAM SEARCH;
                BEGIN
                TYPE KEY STATUS (V(RED), V(GREEN), V(YELLOW));
                TYPE DBASE TABLE (1000);
                        BEGIN
                        ITEM CODE KEY:
                        ITEM VALUE U;
                        END
                TABLE DATA DBASE;
                ITEM CURVAL U;
                GETVALUE (DATA);
                CURVAL = RETRIEVE (V(RED));
                PROC RETRIEVE (ARG1) U;
                        BEGIN
                        ITEM ARG1 KEY;
                        FOR I : 0 BY 1 WHILE I <= 1000;
                                IF CODE (I) = ARG1;
                                        RETRIEVE = VALUE (I);
                        ERROR (20);
                        END
                END
DEF PROC GETVALUE (ARGTAB);
        BEGIN
        TABLE ARGTAB DBASE;
        ...
        END
DEF PROC ERROR (ERRNO);
        BEGIN
        ITEM ERRNO U;
        ...
        END
TERM
```

Figure 1-1. Main-program-module complete with Subroutines

This main-program-module consists of a program-body and two non-nested-subroutines. The program-body contains two type-declarations, a table-declaration, an item-declaration, two executable statements, and a nested subroutine-definition.

SECTION 2

SUBROUTINE-DEFINITION

SUBROUTINE-DEFINITION

A subroutine is either a procedure or a function.

A subroutine-definition defines the subroutine name and the statements to be executed (subroutine-body).

A form of a subroutine-definition for a procedure is:

> PROC name [use-attribute] [(formal-list)] ;
>
> subroutine-body

A form of a subroutine-definition for a function is:

> PROC name [use-attribute] [(formal-list)] item-type-description;
>
> subroutine-body

A simple subroutine-definition contains the word PROC, the name of the procedure or function and the subroutine-body, which consists of declarations, executable statements, and perhaps nested subroutines.

A function-definition is distinguished from a procedure-definition by the presence of an item-type-description. A function returns a result with those attributes defined by the item-type-description.

The square brackets indicate that use-attribute and the parenthesized list of formal parameters can be omitted. Use-attribute indicates whether the subroutine is recursive or reentrant. The compiler uses this attribute to allocate data within the procedure properly. Use-attribute and the parameters are described in detail in subsequent sections.

The subroutine-body consists of a BEGIN/END pair surrounding local declarations, at least one executable statement and definitions of any local subroutines used in the subroutine-body.

Figure 1-1 contains two examples of procedure subroutine-definions, GETVALUE and ERROR, and one function subroutine definition, RETRIEVE. GETVALUE has one formal-parameter, ARGTAB of type DBASE. ERROR has one formal-parameter, ERRNO. The function RETRIEVE returns a result of type U BITSINWORD-1 associated with the name RETRIEVE. (BITSINWORD-1 is the default size if the integer size is omitted from the item-type-description.)

SECTION 3

SUBROUTINE INVOCATION

SUBROUTINE INVOCATION

At some point, if a subroutine is to be executed, it must be invoked. This is done by calling the subroutine.

If the subroutine is a procedure, the subroutine is invoked by a procedure-call-statement. The form is:

procedure-name [actual-parameter-list] [abort-phrase];

The abort-phrase is discussed in Section 9.

If the subroutine is a function, the subroutine is invoked in a formula by a function-call. The form is:

function-name [actual-parameter-list]

Control never passes through a subroutine-definition unless the subroutine is called in a statement.

Figure 1-1 contains three calling statements for subroutines. GETVALUE (DATA) is a procedure-call-statement which invokes the procedure GETVALUE and passes the actual-parameter DATA. Note that the formal-parameter ARGTAB and the actual-parameter DATA have the same table-type-description. RETRIEVE (V(RED)) is a function-call and ERROR(20) is another example of a procedure-call-statement.

The attributes of the formal-parameter-list and the actual-parameter-list are discussed in Section 5.

SECTION 4

SUBROUTINE TERMINATION

SOFTECH

SUBROUTINE TERMINATION

Subroutines may be terminated normally or abnormally. A normal termination of a subroutine means all value-result parameters and the function-return value will be set (see Section 7). A subroutine may be terminated normally by:

- execution of the last executable statement of a subroutine-body
- execution of a return-statement

An abnormal termination of a subroutine means the value-result parameters and the function-return value will not be set. Output parameters passed by reference may be partially set. Abnormal terminations are discussed in Section 9.

Consider the following example:

Procedure Definition;	Procedure Call
PROC TRIAL1;	statement
BEGIN	.
declarations	.
executable statements	.
END	TRIAL1;
	.
	.
	.

When the call to TRIAL1 is encountered, the statements inside TRIAL1 will be executed in succession. When the END is encountered, control will be transferred to the statement following the subroutine call.

RETURN-STATEMENT

The return-statement is used to effect a normal return from a subroutine. When a return-statement is executed, the execution of the subroutine is terminated, any parameters that have value-result binding are set, and control returns to the point following the subroutine-call.

The form of a return-statement is:

```
RETURN ;
```

Suppose the programmer wants to search for a particular character string in a table of character strings, wants to stop the search either when the desired character string is found or when the end of the table is reached. A return-statement for the case in which the character string match is found can be used as follows:

```
PROC SEARCH(TABNAME,STRING:POSITION);
    BEGIN
    TABLE TABNAM(999);
        ITEM TABSTRING C 10;
    ITEM STRING C 10;
    ITEM POSITION U;
    FOR POSITION : 0 BY 1 WHILE POSITION < 1000;
        IF TABSTRING(POSITION) = STRING;
            RETURN;
    NOTFOUND(STRING);
    END
```

If the character string STRING is found in the table TABNAME, the RETURN is executed and the output parameter POSITION gives the entry number in the table where the match occurred. If the character string is not found, the procedure NOTFOUND is called and the output parameter POSITION contains the value 1000.

A return-statement causes a return only from the subroutine in which it is given, not from any subroutines in which the subroutine containing the return is nested.

SECTION 5

PARAMETERIZED SUBROUTINES

PARAMETERIZED SUBROUTINES

A subroutine may be used to perform one set of actions on several different sets of data. That data may be passed into the subroutine as parameters.

The form of a parameterized subroutine-definition for a procedure is:

```
PROC name ( formal-list ) ;
    subroutine-body
```

The form of a parameterized subroutine-definition for a function is:

```
PROC name ( formal-list ) item-type-description ;
    subroutine-body
```

The formal-list specifies the formal parameters. The form of the formal-list of parameters is:

```
[ input-formal, ... ] [ : output-formal, ... ]
```

A subroutine may have no formal parameters, just input-formals, just output-formals, or both input- and output-formals.

The names of formal parameters in a formal-list must be unique.

Examples

```
PROC SAMPLE'PROC (SAMPL'IN : SAMPL'OUT);
PROC ALPHABET;
PROC CHANGE ( : AMOUNT, COINST'TAB);
PROC SAMPLE'FUNC (SAMPL'IN) F;
PROC PRIME (NUMBER : FACTAB) B 1;
```

Function Definition and Call -- Example

```
PROC SAMPLE'FUNC (SAMPL'IN)F;
  BEGIN
  ITEM SAMPL'IN F;
  SAMPLE'FUNC = (SAMPL'IN ** 2)/
      2.14;
  END
```

This is a function because it is defined with an item-type-description. It has an input parameter.

```
statement
.
.
.
NUMBER = SAMPLE'FUNC (COUNT);
.
.
.
```

This is a function-call. When this is encountered, the value of COUNT is copied into SAMPL'IN and the value of SAMPLE'FUNC is assigned in the function. When the function is finished executing, the value of SAMPLE'FUNC is assigned to NUMBER.

Procedure Definition and Call -- Example

```
PROC SAMPLE'PROC (SAMPL'IN :
SAMPL'OUT);
  BEGIN
  ITEM SAMPL'IN F;
  ITEM SAMPL'OUT F;
  SAMPL'OUT + (SAMPL'IN ** 2)/
      2.14;
  END
```

This is a procedure-definition. It has an input parameter and an output parameter.

```
statement
.
.
.
SAMPLE'PROC (COUNT : NUMBER);
.
.
.
```

This is a procedure-call-statement. When this statement is encountered, the value of COUNT is copied in to SAMPL'IN, the value of NUMBER is copied in to SAMPL'OUT. When the procedure is finished executing the value of SAMPL'OUT will be copied into NUMBER.

INPUT AND OUTPUT PARAMETERS

The parameters given in a subroutine-definition are called formal parameters because they represent the parameters for the purpose of defining the computations to be performed by the subroutine using the parameters. The parameters given in a subroutine-call are called actual parameters because they are the parameters for that invocation of the subroutine.

A formal input parameter designates a parameter that receives a value from the corresponding actual parameter. A formal output parameter designates a parameter that receives a value from the corresponding actual parameter when the subroutine is called and returns a value to the corresponding actual when the subroutine is terminated in a normal way. All formal parameters must be declared in the declarations section of the subroutine-body.

If, in the course of the execution of a procedure, a formal parameter is used in a context in which its value can be altered, then it must be declared as an output parameter. That is, the value of a formal input parameter cannot be changed within a subroutine.

```
PROC SAMPLE'PROC ( IN : OUT );
       BEGIN
       ITEM IN S;
       ITEM OUT S;
       :
       IN = 12; "Illegal statement"
       OUT = 12; "Legal statement"
       :
       END
```

The number of input actual parameters in the call must be the same as the number of input formal parameters in the definition. Similarly, the number of output actuals must be the same as the number of output formals.

All parameters must be declared within a subroutine. A formal input parameter can be a data-object, a label, or a subroutine. A formal output parameter can be a data-name only. Labels as input parameters are discussed in Section 8.

A formal parameter cannot be declared to be a constant or a type. Declarations of formal parameters must not contain allocation specifiers or initial values. Formal parameters cannot be declared to be external.

The data type of an actual parameter must match the data type of the corresponding formal parameter. The rules for type matching of actual and formal parameters depend on the data types of the parameters:

Items -- The data type of an actual parameter must be compatible with the data type of the corresponding formal parameter.

Tables -- The data type of the actual and formal parameter must be equivalent. The attributes and allocation order of all components of the table must be equivalent.

Blocks -- The data type of a block actual matches the data type of a block formal if (1) the types and order of the components match exactly, (2) an !ORDER directive is either present or absent in both, and (3) overlay declarations in both blocks have the same effect.

SUBROUTINE DECLARATIONS

If a formal parameter is a subroutine-name, it is declared by a subroutine-declaration. A subroutine-declaration contains the information necessary to describe a call on the subroutine.

The form of a subroutine-declaration is:

```
PROC procedure-name [ use-attribute ] [ ( formal-list ) ]
     type-description;
  parameter-declaration
```

If the subroutine has parameters, then a declaration must be given for each parameter. If the subroutine does not have any parameters, then a null declaration must be given instead of the parameter declarations. No other declarations can be given in a subroutine-declaration. Declarations of local data, as well as the executable statements and any local subroutine-definitions, are not given in a subroutine-declarations; they can appear only in the subroutine-definition.

If the subroutine-declaration includes a type-description, then it declares a function; otherwise, it declares a procedure.

As an example of the use of a subroutine parameter, consider the following:

```
PROC VERIFY(TAB,SUB1,SUB2:SUM);
      BEGIN
      TABLE TAB(*);
            ITEM TABENT F;
      PROC SUB1;
            BEGIN
            END
      PROC SUB2;
            BEGIN
            END
      ITEM SUM F;
      SUM = 0.0;
      FOR I:LBOUND(TAB,0) BY 1 WHILE I <= UBOUND(TAB,0);
            BEGIN
            IF TABENT(I) <THRESHOLD;
                  SUB1;
            SUM = SUM + TABENT(I)**2;
            IF SUM > MAXSUM;
                  SUB2;
            END
```

Suppose you call the procedure VERIFY as follows:

```
VERIFY(NEWDATA,LOWDATA,OVERFLOW:NEWSUM);
```

If an entry within the table NEWDATA is less than THRESHOLD, the procedure LOWDATA is invoked. If SUM is greater than MAXSUM, OVERFLOW is invoked.

In the following example, ANALYSIS invokes the function ROOT'FINDER which in turn calls the function FUNC'OF'Z.

```
PROC ANALYSIS;
BEGIN
↓
ANSWER = ROOT'FINDER (FUNC'OF'Z,      "Find root of FUNC'OF'Z"
  -1.0, 1.0);
↓
END
PROC ROOT'FINDER (FUNC, LOBOUND,
  HIBOUND) F;
      BEGIN
      PROC FUNC (XX) F;              "Find a root of the
          ITEM XX F;                 FUNCTION, given a lower
      ITEM LOBOUND F;                bound and an upper bound
      ITEM HIBOUND F;                inside which a root can be
      TEMP = FUNC(LOBOUND);          found."
      ↓                              "Call to FUNC'OF'Z"
      END
PROC FUNC'OF'Z (ZZ) F;
      BEGIN
      ITEM ZZ F;
      FUNC'OF'Z = ZZ ** 3 + 2. * ZZ
        ** 2 + 18. * ZZ - 36. ;
      END
```

ASTERISK BOUND TABLE DECLARATIONS

A formal parameter that is a table may be declared with an asterisk bounds to allow you to write subroutines that don't depend on the actual bounds of the table.

```
PROC CLEAR (: TABNAME);
      BEGIN
      TABLE TABNAME (*, *);
            ITEM TABENT U;
      FOR I : 0 BY 1 WHILE I <= UBOUND (TABNAME, 0);
            FOR J : 0 BY 1 WHILE J <= UBOUND (TABNAME, 1);
                  TABENT (I, J) = 0;
      END
```

The procedure CLEAR may be called for any two-dimensional table with an entry-description that matches that of the formal parameter. This procedure sets all items in table TABNAME to zero.

The bounds are determined from the actual parameter when the subroutine is called. The bounds are treated as unsigned integers and normalized.

By using the subroutine CLEAR with the following two tables, all items in each table are set to zero.

```
TABLE GRAPH (1 : 10, 2 : 20);
      ITEM POINT U;
TYPE SEASON STATUS
      (V(SPRING), V(SUMMER), V(FALL), V(WINTER));
TABLE WEATHER (88, V(FALL));
      ITEM RAINFALL U;

CLEAR (: GRAPH);
CLEAR (: WEATHER);
```

If one dimension is asterisk bound, all must be. No range is permitted in the declaration such as (* : 15).

EXERCISE ON INPUT AND OUTPUT PARAMETERS

The following program contains nine errors. Locate and specify what the errors are.

```
START PROGRAM ERRORS;
BEGIN
      ITEM AA U 15;
      ITEM BB U 15;

      PROC (AA : FUNCA(BB));

      PROC FUNCA (CC) U 15;
            BEGIN
            ITEM CC U 15;
            FUNCA = 12;
            FUNCB (FUNCA);
            END

      PROC FUNCB (FF) U 15;
            BEGIN
            ITEM FF U 15;
            ITEM GG U 15;
            GG = FUNCB * FF;
            END

      PROC FUNCC (HH) U 15;
            BEGIN
            ITEM HH U 15;
            ITEM FUNCC U 15;
            FUNCC = HH * HH;
            END

      PROC PROCA (INA : OUTA);
            BEGIN
            ITEM INA U 15;
            ITEM OUTA U 15 = 234;
            OUTA = INA;
            END

      PROC PROCB (INA) C 10;
            BEGIN
            ITEM INA C 10;
            END

      PROC SUM (INA, INB : OUTA);
            BEGIN
            ITEM INA STATIC S 16;
            ITEM INB S 16;
            ITEM OUTA S 16;
            INA = INA + 2;
            OUTA = INA + INB;
            END
      END
END
TERM
```

ANSWERS

The following program contains nine errors. Locate and specify what the errors are:

```
START PROGRAM ERRORS;
BEGIN
        ITEM AA U 15;
        ITEM BB U 15;

        PROC (AA : FUNCA(BB));          1. Actual-output-parameter does not
                                           match formal-output parameter.
        PROC FUNCA (CC) U 15;
                BEGIN
                ITEM CC U 15;
                FUNCA = 12;
                FUNCB (FUNCA);          2. Illegal function name as an actual-
                END                        parameter.

        PROC FUNCB (FF) U 15;
                BEGIN
                ITEM FF U 15;
                ITEM GG U 15;
                GG = FUNCB * FF;        3. Illegal use of a function name.
                END                     4. FUNCB not assigned a value to
                                           return.
        PROC FUNCC (HH) U 15;
                BEGIN
                ITEM HH U 15;
                ITEM FUNCC U 15;        5. Illegal declaration of a function-name.
                FUNCC = HH * HH;
                END

        PROC PROCA (INA : OUTA);
                BEGIN
                ITEM INA U 15;
                ITEM OUTA U 15 = 234;   6. Illegal preset of formal-parameter.
                OUTA = INA;
                END

        PROC PROCB (INA) C 10;
                BEGIN
                ITEM INA C 10;
                END                     7. No executable statement present.

        PROC SUM (INA, INB : OUTA);
                BEGIN
                ITEM INA STATIC S 16;   8. Illegal STATIC allocation of formal
                ITEM INB S 16;             parameter.
                ITEM OUTA S 16;
                INA = INA + 2;          9. Illegal alteration of input parameter.
                OUTA = INA + INB;
                END
        END
END
TERM
```

SECTION 6

PARAMETERIZED SUBROUTINE INVOCATION

SOFTECH

PARAMETERIZED SUBROUTINE INVOCATION

If the subroutine is a procedure, it is invoked by a procedure-call-statement. The form is:

procedure-name [(actual-list)] ;

If the subroutine is a function, it is invoked in a formula by a function-call. The form is:

function-name [(actual-list)]

The formal-list must match the actual-list in number, type, and order of the parameters. A procedure is invoked by a procedure-call-statement.

Example

```
statement
.
.
.
SAMPLE'PROC (IN'ARG : OUT'ARG);
XX = ZZ + 4 * OUT'ARG;
statement
.
.
.
```

A function is invoked within a formula by a function-call. It returns a value.

Example

```
statement
.
.
.
XX = ZZ + 4 * SAMPLE'FUNC(IN'ARG);
statement
.
.
.
```

The data type of an actual-input or output parameter must be equivalent or implicitly convertible to the type of its corresponding formal-input or output parameter. In addition, the type of a formal-output parameter must be equivalent or implicitly convertible to the type of its corresponding actual.

An actual-input parameter may be an item, table, block, statement-name, or subroutine-name. An actual-input parameter may also be a formula.

An actual-output parameter may be an item, table, or block only. An actual-output parameter must not be a formula.

NOTE: An actual parameter may be listed as both input and output, but all formal-names must be unique.

SUBROUTINES -- EXERCISES

Given the following declarations and definitions:

```
ITEM AA F;
ITEM COUNT U;
ITEM SPEED F;
TABLE LIST (1 : 20);
       ITEM NUMBER U;
PROC CALCULATE (ARGIN : ARGOUT);
       BEGIN
       ITEM ARGIN U;
       ITEM ARGOUT F;
       ...
       END
PROC COMPUTE (VARIN) F;
       BEGIN
       ITEM VARIN U;
       ...
       END
```

Determine whether the following subroutine-call-statements are correct or incorrect.

Subroutine-call-statement	C	I
1. CALCULATE (COUNT + 2 : SPEED);		
2. CALCULATE (COUNT * 5, SPEED);		
3. CALCULATE (COUNT : F(COUNT));		
4. AA = 3. + COMPUTE(COUNT);		
5. CALCULATE (U(SPEED) : SPEED);		
6. CALCULATE (: COUNT);		
7. COMPUTE (3 * COUNT : SPEED);		

ANSWERS

Given the following declarations and definitions:

```
ITEM AA F;
ITEM COUNT U;
ITEM SPEED F;
TABLE LIST (1 : 20);
      ITEM NUMBER U;
PROC CALCULATE (ARGIN : ARGOUT);
      BEGIN
      ITEM ARGIN U;
      ITEM ARGOUT F;
      ...
      END
PROC COMPUTE (VARIN) F;
      BEGIN
      ITEM VARIN U;
      ...
      END
```

Determine whether the following subroutine-call-statements are correct or incorrect.

Subroutine-call-statement	C	I
1. CALCULATE (COUNT + 2 : SPEED);	X	
2. CALCULATE (COUNT * 5, SPEED);		X (need 1 in, 1 out)
3. CALCULATE (COUNT : F(COUNT));		X (no formulae for out)
4. AA = 3. + COMPUTE(COUNT);	X	
5. CALCULATE (U(SPEED) : SPEED);	X	
6. CALCULATE (: COUNT);		X (need 1 in 1 out; output is type F)
7. COMPUTE (3 * COUNT : SPEED);		X(need function-call; only 1 in, zero out)

SECTION 7

PARAMETER BINDING

PARAMETER BINDING

BINDING

The way in which a formal parameter is bound depends on its type and input/output status. JOVIAL (J73) uses three types of binding: value, value-result, and reference.

A formal input parameter that is an item is bound by value. A formal output parameter that is an item is bound by value-result. A formal parameter that is a table or block is bound by reference.

For all three types of binding, actual parameter values or the location of actual parameter values are evaluated when the subroutine is invoked and are not reevaluated while the subroutine is being executed.

VALUE BINDING

If a formal parameter is bound by value, it denotes a distinct object of the type specified in the formal parameter declaration. When the subroutine is called, the value of the actual parameter is assigned to that object.

For example, given the following procedure-declaration:

```
PROC RUNTIMER(ARG1);
      BEGIN
      ITEM ARG1 U;
      FOR I:0 BY 1 WHILE I < ARG1**2;
            CORRELATE(ARG1,I);
      END
```

And the programmer calls the procedure:

```
RUNTIMER(CLOCK1);
```

The formal parameter ARG1 is assigned the value of CLOCK 1 when the procedure is called.

VALUE-RESULT BINDING

If a formal parameter is bound by value-result, it denotes a distinct object of the type specified in the formal parameter declaration to which the value of the actual is assigned when the subroutine is called. In addition, when the subroutine is exited normally, the value of the formal is assigned to the corresponding actual. If the subroutine is terminated in an abnormal way, the value of the formal is not assigned to the actual. Abnormal returns from subroutines are discussed in Section 9.

Given the following procedure:

```
PROC MINMAX(VECTOR:MIN,MAX);
    BEGIN
    TABLE VECTOR(99);
        ITEM V1 U;
    ITEM MIN U;
    ITEM MAX U;
    MIN, MAX = V1(0);
    FOR I : 1 BY 1 WHILE I <= 99;
        BEGIN
        IF V1(I) < MIN;
            MIN = V1(I);
        IF V1(I) > MAX;
            MAX = V1(I);
        END
    END
```

And this procedure call:

```
MINMAX(RETURNS:RMIN,RMAX);
```

The procedure MINMAX finds the minimum and maximum values in the table RETURNS and, on completion, sets the value of RMIN to the minimum value (MIN) and RMAX to the maximum value (MAX).

REFERENCE BINDING

If a formal parameter is bound by reference, the actual parameter and the formal parameter denote the same physical object. Any change in the formal parameter entails an immediate change in the value of the actual parameter.

Suppose the programmer wants to square the items of a table and then calculate the sum of the pairwise quotients. The item SIZE gives the number of entries in the table currently in use. SIZE is always an even number. The following can be written:

```
PROC MEAN(:ARGBLOCK);
        BEGIN
        BLOCK ARGBLOCK
                BEGIN
                ITEM SIZE U;
                ITEM SUM U;
                TABLE ARGTAB(1:1000);
                        ITEM VALUE S;
                END
        SUM = 0;
        FOR I : 1 BY 2 WHILE I < SIZE;
                BEGIN
                IF VALUE(I+1) = 0;
                        ABORT;
                VALUE(I) = VALUE(I)**2;
                VALUE(I+1) = VALUE(I+1)**2;
                SUM = SUM + VALUE(I)/VALUE(I+1);
                END
        END
```

Suppose STATISTICS is a block that is declared as follows:

```
BLOCK STATISTICS;
        BEGIN
        ITEM STATSIZE U = 10;
        ITEM STATSUM U;
        TABLE STATTAB(999);
                ITEM STATVALUE S = 2,4,3,4,8,6,9,0,11,3;
        END
```

Suppose a call is made to the procedure MEAN with the actual parameter STATISTICS, as follows:

```
MEAN(:STATISTICS);
```

The block STATISTICS is bound by reference to the formal parameter ARGBLOCK. Each change to SUM results in an immediate change to STATSUM. Similarly, a change in VALUE(I) results in a change in STATVALUE(I). If the procedure terminates abnormally as a result of finding a zero value in the table, STATSUM has the value computed up to that point and the values of the table STATTAB are changed up to the point at which the zero quotient was encountered.

NORMAL AND ABNORMAL TERMINATION

Subroutines may end normally, as discussed in Section 4 or abnormally (Section 9).

Normal termination means that value-result parameters are copied out, parameters bound by reference are fully set, and the function return-value is copied out.

NORMAL TERMINATION	PARAMETERS
VALUE-RESULT	copied out
REFERENCE	set fully
RETURN-VALUE	copied out

Abnormal termination means that value-result parameters are not copied out, parameters bound by reference are set only until the point of abnormal termination, and the function return-value is not copied out.

ABNORMAL TERMINATION	PARAMETERS
VALUE-RESULT	not copied out
REFERENCE	partially set
RETURN-VALUE	not copied out

PARAMETER BINDING -- EXERCISES

Given the following declarations:

```
ITEM COUNT U;
ITEM SPEED F;
TABLE STOCKS (1 : 100);
      BEGIN
      ITEM NAME C 10;
      ITEM QUOTE C 6;
      ...
      END
```

Name the kind of binding for each parameter and describe what occurs:

<u>Subroutine-calls</u>

1. CALCULATE (COUNT : SPEED);

2. COMPUTE (STOCKS);

3. GETRESULT (: STOCKS);

4. RESULT (NAME(10));

5. PROCESS (COUNT ** 2);

6. ANSWER (: QUOTE (4));

> *NOTE: Since tables and blocks are bound by reference, the overhead of copying a long character string is saved by declaring it in a table or block and passing the table or block as a parameter.*

ANSWERS

Given the following declarations:

```
ITEM COUNT U;
ITEM SPEED F;
TABLE STOCKS (1 : 100);
        BEGIN
        ITEM NAME C 10;
        ITEM QUOTE C 6;
        ...
        END
```

Name the kind of binding for each parameter and describe what occurs:

Subroutine-calls

1. CALCULATE (COUNT : SPEED);	COUNT - value SPEED - value-result
2. COMPUTE (STOCKS);	STOCKS - reference
3. GETRESULT (: STOCKS);	STOCKS - reference
4. RESULT (NAME(10));	NAME(10) - value
5. PROCESS (COUNT ** 2);	COUNT**2 - value
6. ANSWER (: QUOTE (4));	QUOTE(4) - value-result

SECTION 8

USE-ATTRIBUTE

SOFTECH

USE-ATTRIBUTE

A recursive subroutine is one that calls itself either directly or indirectly.

A reentrant subroutine is one that may be used by concurrent processes.

A programmer must specify if a subroutine is to be used in either a recursive or reentrant way. The default is neither. The form is:

```
PROC name [ use-attribute ] [ ( formal-list ) ] [ item-type-description ] ;
        subroutine-body
```

The form of the use-attribute is:

```
REC | RENT
```

for recursive or reentrant procedures respectively.

Example

```
PROC RFACT REC (ARG) U;
        subroutine-body
```

REENTRANT SUBROUTINES

A reentrant subroutine is one that can be called from several different tasks. If the compiler knows the maximum number of separate tasks that can invoke a reentrant subroutine in a given system, it can allocate storage for the subroutine statically. In general, the compiler dynamically allocates storage for reentrant subroutines. Locally-declared STATIC data is shared among all re-entrant invocations.

Consider the following example:

```
START PROGRAM REENT;
      BEGIN
      ITEM SA S 12;
      ITEM SB S 8;
      ITEM GLOB S 12;

      SA = 0;   SB = 12;   GLOB = 0;

      SA = REENA (SB:GLOB);
      SA = REENA(SB:GLOB); "CALL TWICE TO TEST STATIC ALLOCATION"
      IF SA = 144 AND GLOB = 24;
            "RE-ENTRANT FUNCTION WITH STATIC ALLOCATION
            AND IMPLICITY RETURN IS CORRECT"
            SB=12;

      PROC REENA RENT (INA : OUTB) S 8;
            BEGIN
            ITEM INA S 8;
            ITEM OUTB S 7;
            ITEM GLOBC STATIC S 8 = 0;
            GLOBC = INA + GLOBC;
            OUTB = GLOBC;
            REENA = INA * INA;
            END
      END
TERM
```

RECURSIVE SUBROUTINES

```
      PROC RFACT REC (ARG) U;
            BEGIN
            ITEM ARG U;
            IF ARG <= 1;
                  RFACT = 1;
            ELSE
                  RFACT = RFACT(ARG-1) * ARG;
            END
```

A call on this function with an argument of 6 produces six calls. The calls can be diagrammed as follows:

```
                              RFACT(6)

      First call                  RFACT(5) * 6

      Second call                 (RFACT(4) * 5) * 6

      Third call                  ((RFACT(3) * 4) * 5) * 6
```

Fourth call	(((RFACT(2) * 3) * 4) * 5) * 6
Fifth call	((((RFACT(1) * 2) * 3) * 4) * 5) * 6
Sixth call	(((((1) * 2) * 3) * 4) * 5) * 6
	= 720

RFACT illustrates the use of recursion clearly. In practice, however, a function like this is not written recursively because the computation is too simple to justify the overhead associated with the repetitive function calling mechanism. In the above example, dynamically allocated memory is required for every integer from 1 to the value of ARG, since there is a separate function call for each of these values.

The function RFACT is obviously recursive because it calls itself. Some subroutines are less obviously recursive because they call other subroutines that, in turn, call them. A recursive subroutine is also re-entrant.

The factorial computation could be done iteratively, as follows:

```
PROC IFACT (ARG) U;
       BEGIN
       ITEM ARG U;
       ITEM TEMP U;
       TEMP = 1;
       FOR I : ARG BY -1 WHILE I > 1;
              TEMP = TEMP * 1;
       IFACT = TEMP;
       END
```

Or, it could be done by a table look-up method.

If REC is present, physical allocation of locally-declared automatic data will occur dynamically. The data will be allocated and deallocated when the subroutine is entered and exited, respectively. This assures that separate copies of the local data will exist for each successive call in the recursive chain. Locally-declared STATIC data, however, will be allocated once, and the same storage will be used for all calls of that subroutine throughout the complete-program.

SECTION 9

ABNORMAL SUBROUTINE TERMINATION

SOFTECH

ABNORMAL SUBROUTINE TERMINATION

An abnormal termination of a subroutine means the value-result parameters and the function-return value will not be set (see Section 7). Output parameters passed by reference may be partially set.

A subroutine may be terminated abnormally by:

- a goto-statement to a formal-input statement-name
- a stop-statement
- an abort-statement

STATEMENT-NAME DECLARATIONS

If a formal-parameter is a statement-name, it is declared by a statement-name declaration. The form of a statement-name declaration is:

LABEL statement-name ,... ;

A GOTO statement to a label that is a formal parameter results in the subroutine being exited and control being sent to the label that is supplied as the actual parameter.

Statement-name parameters are useful for subroutines that have more than one possible error exit. For example, consider the following subroutine:

```
PROC VERIFY(TAB,L1,L2:SUM);
     BEGIN
     TABLE TAB(*);
          ITEM TABENT F;
     LABEL L1,L2;
     ITEM SUM F;
     SUM = 0.0;
     FOR I : LBOUND(TAB,0) BY 1 WHILE I <= UBOUND(TAB,0);
          BEGIN
          IF TABENT(I) < THRESHOLD;
               GOTO L1;
          SUM = SUM + TABENT(I)**2
          IF SUM > MAXSUM;
               GOTO L2;
          END
     END
```

NOTE: *This is the only case when labels must be declared.*

Suppose the programmer calls the procedure VERIFY as follows:

```
VERIFY(NEWDATA,ERROR1,ERROR5:NEWSUM);
```

The procedure VERIFY is terminated abnormally under two separate conditions. If an entry in NEWDATA is less than THRESHOLD, the procedure is terminated abnormally and control is sent to the label ERROR1. If SUM is greater than MAXSUM, the procedure is terminated abnormally and control is sent to the label ERROR5.

The use of a statement label formal parameter to exit a subroutine constitutes an abnormal termination from the subroutine.

Statement-Names as Parameters -- Examples

```
PROC RANGE (VELOCITY,L1, L2) S;
       BEGIN
       ITEM VELOCITY S;
       LABEL L1, L2;
       RANGE = 0;
       CASE VELOCITY;
              BEGIN
              (DEFAULT) : GOTO L1;
              (0 : 100) : RANGE = VELOCITY;
              (101 : 199) : GOTO L2;
              END
       END
```

The subroutine could be used in the following way:

```
       FOR I : 1 BY 1 WHILE I <= 1000;
              BEGIN
              .
              .  NOTE = 'WITHIN ACCEPTABLE VALUES';
              .
        XX =  RANGE(SPEED, ERROR1, ERROR2);
              EXIT;
ERROR1:       NOTE = 'BELOW ACCEPTABLE VALUES';
              EXIT;
ERROR2:       NOTE = 'ABOVE ACCEPTABLE VALUES';
              END
```

STOP-STATEMENT

The stop-statement causes the execution of the program to stop. The form is:

STOP [stop-code] ;

The value of stop-code is supplied to the operating system and interpreted in a machine-dependent manner.

A stop-statement may be used anywhere within a program. When it is used in a subroutine, value-result parameters and the function-return value are not set. Parameters passed by reference may be partially set.

Stop-code is an integer formula that may take on the following value defined using implementation parameters:

MINSTOP <= stop-code <= MAXSTOP

ABORT-STATEMENT AND ABORT-PHRASE

The abort-statement is of the form:

ABORT ;

The abort-statement within a subroutine works with the abort-phrase on a procedure-call-statement to produce an abnormal exit from a procedure and transfer control to another statement. It is mostly used for error processing.

If an abort-statement is executed, value-result parameters are not set.

A procedure-call-statement may have an abort-phrase:

procedure-name [(actual-list)] [abort-phrase] ;

The form of the abort-phrase is:

ABORT statement-name

NOTE: A function-call does not have an abort-phrase.

When an abort-statement is executed, the subroutine is terminated. If the call to the subroutine has an abort-phrase, control is transferred to the statement-name given in that phrase. If the call to the subroutine does not have an abort-phrase, the calling subroutine is terminated abnormally similar to STOP.

<u>**Abort-Statement -- Example**</u>

```
BEGIN
.
.
.
CALCULATE (LENGTH : ESTIMATE) ABORT RECOVERY;
.
.
.
RECOVERY : STOP;
.
.
.
END

PROC CALCULATE (IN : OUT);
       BEGIN
       .
       .
       .
       IF IN < 0;
              ABORT;
       OUT = IN * 3.2;
       .
       .
       .
       END
```

SECTION 10

SUMMARY

SOFTECH

SUMMARY

A subroutine may be used to add structure to a program or to repeat a series of actions at several points in a program. A subroutine may be a procedure or a function. A subroutine-definition defines a subroutine. The form is:

PROC name [REC | RENT] [(formal-list)] [item-type-description] ;

subroutine-body

If the item-type-description is present, the subroutine is a function. Otherwise, it is a procedure.

Use-attribute has the form:

REC | RENT

If a subroutine is used in a recursive or reentrant way, it must be declared with the appropriate use-attribute.

The formal-list has the form:

{ input-formal, ...
[input-formal, ...] : output-formal, ... }

Input-formals may be:

{ item-name | table-name | block-name
statement-name | subroutine-name }

Output-formals may be:

{ item-name | table-name | block-name }

NOTE: A formal parameter may not be named both as an input- and output-formal but an actual parameter may.

A procedure is invoked with a procedure-call-statement. The form is:

procedure-name [(actual-list)] [abort-phrase] ;

A function is invoked in a formula with a function-call. The form is:

function-name [(actual-list)]

The actual-list has the form:

$$\left\{ \begin{array}{l} \text{input-actual, ...} \\ \text{[input-actual, ...] : output-actual, ...} \end{array} \right\}$$

Input-actuals may be:

$$\left\{ \begin{array}{l} \text{item-name} \mid \text{table-name} \mid \text{block-name} \\ \text{statement-name} \mid \text{subroutine-name} \end{array} \right\}$$

Output-actuals may be:

$$\left\{ \text{item-name} \mid \text{table-name} \mid \text{block-name} \right\}$$

The number, type, and order of the actual parameters must match the formal parameters. An input-actual may be a formula, an output-actual *must not*.

Parameter Binding:

An item input-actual parameter is bound by value to its corresponding formal parameter (copy-in).

An item output-actual parameter is bound by value-result to its corresponding formal parameter (copy-in, copy-out)

A table or block input- or output-actual is bound by reference to its corresponding formal parameter.

All formal parameters must be declared in a subroutine-definition.

A table may be declared with asterisk bounds. The actual bounds are treated as integers and begin at zero.

The form of a statement-name declaration is:

LABEL statement-name, ... ;

The form of a subroutine-declaration is:

PROC NAME [use-attribute] [(formal-list) [item-type-description] ;

parameter-declaration

All formal parameters must be declared in the parameter-declaration. If there are no parameters, a null-declaration must be given.

If a subroutine is terminated normally, all value-result parameters and parameters passed by reference are set. A subroutine may be terminated normally by:

- the execution of the last statement
- a return-statement

A return-statement returns control to the point of the call. The form of the return-statement is:

RETURN ;

If a subroutine is terminated abnormally, value-result parameters will not be set and parameters passed by reference may be partially set. A subroutine may be terminated abnormally by:

- a goto-statement to a formal-input statement-name
- a stop-statement
- an abort-statement

A stop-statement stops execution of the program. The form of a stop-statement is:

STOP [stop-code] ;

The form of an abort-statement is:

ABORT ;

The form of the abort-phrase is:

ABORT statement-name

The abort-phrase appears on a procedure-call-statement. If an abort-statement is executed, that procedure is terminated, control returns to the point of call, and control is transferred to the statement labelled by the statement-name in the abort-phrase. If there is no abort-phrase, that procedure is terminated, control returns to the point of call, and control is transferred to the statement labelled by the statement-name in the abort-phrase. This process continues until an abort-phrase is encountered or the program is terminated.

THE JOVIAL (J73) WORKBOOK

VOLUME 7

BUILT-IN FUNCTIONS

1081-1

April 1981

Submitted to

Department of the Air Force
Rome Air Development Center
I S I S
Griffiss Air Force Base, NY 13441

Prepared by

SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02154

PREFACE

This workbook is intended for use with Tape 7 of the JOVIAL (J73) Video Course. Its purpose is to elaborate upon and reinforce concepts and language rules introduced in the videotape.

This workbook addresses 17 built-in functions which are predefined by the language and may be called from anywhere in a JOVIAL (J73) program. The final section is a summary of the information presented in this segment.

TABLE OF CONTENTS

SYNTAX

The syntax conventions used in the JOVIAL (J73) Video Course and workbook are as follows:

Syntax	Meaning	Sample Expansions
[some-feature]	Brackets indicate an optional feature.	some-feature OR nothing
{one | other}	Braces with a vertical bar indicate disjunction-a choice between alternatives.	one OR other
{this-one that-one}	Braces with each feature on separate lines indicate disjunction - a choice between alternatives.	this-one OR that-one
letter ...	The sequence '...' indicates one or more repetitions of a feature.	letter letter letter letter letter letter ...
(letter),...	The sequence "..." following a comma (or a colon) indicates one or more repetitions of the feature separated by commas (or colons).	(letter) (letter) (letter) (letter) (letter) (letter) ...
[{this-one that-one}] + another	Syntax symbols may be combined.	this-one + (another) that-one + (another) + (another)

SECTION 1

THE LOC FUNCTION

SOFTECH

THE LOC FUNCTION

The LOC function returns a pointer to the machine address of its argument.

It has the form:

LOC (argument)

The argument of the LOC function can be a data object name, a statement-name, or a subroutine-name. The LOC function returns a pointer value. If the argument is declared using a type-name, the LOC function returns a typed pointer whose type-name is the type given in the declaration. Otherwise the LOC function returns an untyped pointer.

The LOC function is used most often to obtain a value for a pointer to be used in a pointer-qualified reference.

If the argument of a LOC function is a statement-name or a subroutine-name, the LOC function returns the machine address used to access the designated statement or subroutine. The LOC function may not be used to obtain the address of a built-in function.

The LOC of a subroutine whose name appears in an inline-declaration or of a statement-name within an inline subroutine is implementation defined.

The LOC Function -- Examples

Given the declarations:

```
TYPE GRID TABLE;
    BEGIN
    ITEM XCOORD U;
    ITEM YCOORD U;
    END
```

```
TABLE BOARD(20,10) GRID;

TYPE SQUARE TABLE(20,10) GRID;

TABLE CHECK SQUARE;
```

Function Call	Result
LOC(BOARD)	An untyped pointer that contains the address of BOARD.
LOC(BOARD(3,7))	A pointer of type GRID that contains the address of the specified entry of BOARD.
LOC(CHECK)	A pointer of type SQUARE that contains the address of the table CHECK.

THE LOC FUNCTION -- EXERCISES

Given the declarations:

```
TYPE GRID TABLE:
        BEGIN
        ITEM XCOORD U;
        ITEM YCOORD U;
        END
TABLE BOARD(20,10) GRID;
TYPE SQUARE TABLE(20,10) GRID;
TABLE CHECK SQUARE.
```

Are the following pointer-qualified references correct or incorrect?

Reference	Correct	Incorrect
XCOORD(3,7) @ (LOC(BOARD))		
XCOORD @ (LOC(BOARD(3,7)))		
XCOORD(3,7) @ (LOC(CHECK))		

ANSWERS

Given the declarations:

```
TYPE GRID TABLE;
        BEGIN
        ITEM XCOORD U;
        ITEM YCOORD U;
        END
TABLE BOARD(20,10) GRID;
TYPE SQUARE TABLE(20,10) GRID;
TABLE CHECK SQUARE;
```

Are the following pointer-qualified references correct or incorrect?

Reference	Correct	Incorrect
XCOORD(3,7) @ (LOC(BOARD))		X (LOC(BOARD) is untyped, cannot dereference untyped pointer)
XCOORD @ (LOC(BOARD(3,7)))	X	
XCOORD(3,7) @ (LOC(CHECK))	X	

SECTION 2

THE ABS FUNCTION

SOFTECH

THE ABS FUNCTION

The ABS function produces a value that is the absolute value of its argument.

The form is:

ABS (numeric-formula)

The result is equal to -(numeric-formula) if the numeric-formula is negative and equivalent to +(numeric-formula) otherwise. The type of the result is the type of the numeric formula preceded by prefix operator. For example, the type of the absolute value of a single-word unsigned integer formula is a single-word signed integer.

The type of the absolute value of a floating point formula is the type of the larger precision operand.

<u>Examples</u>

Given the following item-declarations:

ITEM TIME U 5 = 2;

ITEM VELOCITY F = 2.356;

ITEM RANGE S 20 = 26;

If BITSINWORD is 16 and the items have their preset values at the time the function-calls are executed, ABS produces the following results:

Function call	Function Value	Type
ABS(RANGE)	25	S 31
ABS(TIME)	2	S 15
ABS(RANGE/TIME)	13	

EXERCISES

Given the following declarations:

ITEM LENGTH U = 20;

ITEM HEIGHT U = 15;

Assuming the items have their preset values and BITSINWORD = 16, give the result and type of the following function calls:

Function Call	Result	Type
ABS(LENGTH)		
ABS(HEIGHT-LENGTH)		

ANSWERS

Given the following declarations:

ITEM LENGTH U = 20;

ITEM HEIGHT U = 15;

Assuming the items have their preset values and BITSINWORD = 16, give the result and type of the following function calls:

Function Call	Result	Type
ABS(LENGTH)	20	S 15
ABS(HEIGHT - LENGTH)	5	S 15

SECTION 3

THE SGN FUNCTION

THE SGN FUNCTION

The SGN function returns an integer that tells whether the value of the numeric-formula is negative, positive or zero.

The form is:

SGN (numeric-formula)

The result is a signed one-bit integer (S 1) that is 1 if the value is positive, 0 if zero, and -1 if negative.

<u>Examples</u>

Given the following item-declarations:

```
ITEM TIME U 5 = 2;

ITEM VELOCITY F = 2.356;

ITEM RANGE S 20 = -25;
```

If the items have their preset values at the time the function-calls are executed, the following results:

Function call	Function value
SGN(VELOCITY)	+1
SGN(RANGE)	-1
SGN(TIME/RANGE)	0 or -1 (implementation dependent truncation)

SGN FUNCTIONS -- EXERCISES

Given the following declarations:

ITEM LENGTH U = 20;

ITEM HEIGHT U = 15;

ITEM WIDTH U = 5;

Assuming the items have their preset values and BITSINWORD = 16, give the result and type of the following function calls:

Function call	Result	Type
SGN(LENGTH)		
SGN(HEIGHT - LENGTH)		
SGN(LENGTH - 4 * WIDTH)		

ANSWERS

Given the following declarations:

ITEM LENGTH U = 20;

ITEM HEIGHT U = 15;

ITEM WIDTH U = 5;

Assuming the items have their preset values and BITSINWORD = 16, give the results and type of the following function calls:

Function call	Result	Type
SGN(LENGTH)	+1	S 1
SGN(HEIGHT - LENGTH)	-1	S 1
SGN(LENGTH - 4 * WIDTH)	0	S1

SECTION 4

THE BIT FUNCTION

SOFTECH

THE BIT FUNCTION

The BIT function selects a substring from a bit-formula.

The form is:

BIT (bit-formula , first-bit , length)

The BIT function returns a string of bits with a size attribute equal to the size attribute of the bit-formula argument. Padding, if necessary, is done on the left.

First-bit and length are integer-formulae. First-bit indicates the bit at which the substring to be extracted starts. Length specifies the number of bits in the substring. Bits in a bit string are numbered from left to right, beginning with zero. Length must be greater than zero. The sum of first-bit and length must not exceed the length of the bit-formula.

Example

```
ITEM MASK B 6 = 1B'011010';
```

Function call	Returns
BIT(MASK,4,1)	1B'000001'
BIT(MASK,2,4)	1B'001010'

> *NOTE: Bits are numbered left to right starting with zero.*

THE BIT FUNCTION -- EXERCISES

Given the following declarations:

```
ITEM MASK B 10 = 1B'0011100111';
ITEM FLAGS B 6 = 3B'77';
ITEM REPRESENTATION B 16 = 4B'FFFF';
```

Assuming the declared items have their initialized values, give the results and types of the following function calls:

Function call	Result	Type
BIT(MASK,2,2)		
BIT(FLAGS,1,3)		
BIT(REPRESENTATION,1,12)		

ANSWERS

Given the following declarations:

ITEM MASK B 10 = 1B'0011100111';

ITEM FLAGS B 6 = 3B'77';

ITEM REPRESENTATION B 16 = 4B'FFFF';

Assuming the declared items have their initialized values, give the results and types of the following function calls:

Function calls	Result	Type
BIT(MASK,2,2)	1B'0000000011'	B 10
BIT(FLAGS,1,3)	1B'000111'	B 6 (or 3B'070')
BIT(REPRESENTATION,1,12)	1B'0000111111111111'	B 16 (or 4B'OFFF')

SECTION 5

THE BYTE FUNCTION

SOFTECH

THE BYTE FUNCTION

The BYTE function selects a substring from a character formula. Its form is:

BYTE (character-formula , first-character , length)

The BYTE function returns a string of bytes with a size attribute equal to the size attribute of the character formula argument. Padding, if necessary, is done on the right. Characters are numbered left to right starting with zero. First-character and length are integer formulae.

Example

```
ITEM WORK C 7 = 'SOFTECH';
```

Function call	Returns	Type
BYTE(WORK,0,4)	'SOFT '	C 7
BYTE(WORK,6,1)	'H '	C 7

The value of a byte function may be assigned to the value of any character-item.

For example:

```
ITEM LETTER C 1;
LETTER = BYTE(WORK,6,1);
```

The value of LETTER is the single character that was selected from WORK by using the BYTE function.

THE BYTE FUNCTION -- EXERCISES

Given the declarations:

ITEM NAME C 6 = 'JOVIAL';

ITEM ID C 9 = 'PROG LANG';

Assuming the items have their initialized values, give the results and types of the following function calls.

Function call	Result	Type
BYTE(NAME,3,1)		
BYTE(ID,6,3)		

ANSWERS

Given the declarations:

ITEM NAME C 6 = 'JOVIAL';

ITEM ID C 9 = 'PROG LANG';

Assuming the items have their initialized values, give the results and types of the following function calls.

Function call	Result	Type
BYTE(NAME,3,1)	'I '	C 6
BYTE(ID,6,3)	'ANG '	C 9

SECTION 6

THE BIT AND BYTE PSEUDO-VARIABLES

BIT AND BYTE PSEUDO-VARIABLES

A pseudo-variable is a function that may be used as a target and be assigned a value. The BIT and the BYTE function may be used as pseudo-variables.

Examples

1) ITEM FLAGS B 3 = 1B'111';

The assignment-statement:

```
BIT(FLAGS,0,1) = 1B'0';
```

changes the value of FLAGS from 1B'111' to 1B'011'.

That is, it changes the first bit in the value of the item.

2) ITEM SPORT C 8 = 'SWIMMING';

The assignment-statement:

```
BYTE (SPORT, 1, 7) = 'KIING  ';
```

changes the value of SPORT from 'SWIMMING' to 'SKIING '.

> *NOTES: The BIT pseudo-variable must be assigned a value of type B. The BYTE pseudo-variable must be assigned a value of type C.*

THE BIT AND BYTE PSEUDO-VARIABLES -- EXERCISES

Given the following declarations:

ITEM MASK B 10 = 1B'1111111111';

ITEM DESCRIPTION C 21 = 'THIS IS A DESCRIPTION';

ITEM STATE C 13 = 'MASSACHUSETTS';

Assuming the items have their initialized values, give the value of the item after execution of the statements with the pseudo-variables:

Statement	Value
BIT(MASK,1,1) = 1B'0';	
BYTE(DESCRIPTION,10,10) = 'CODE';	
BYTE(STATE,4,9) = '.';	

ANSWERS

Given the following declarations:

ITEM MASK B 10 = 1B'1111111111';

ITEM DESCRIPTION C 21 = 'THIS IS A DESCRIPTION';

ITEM STATE C 13 = 'MASSACHUSETTS';

Assuming the items have their initialized values, give the value of the item after execution of the statements with the pseudo-variables:

Statement	Value
BIT(MASK,1,1) = 1B'0';	1B'1011111111'
BYTE(DESCRIPTION,10,10) = 'CODE';	'THIS IS A CODE N'
BYTE(STATE,4,9) = '.';	'MASS. '

SECTION 7

THE REP FUNCTION

THE REP FUNCTION

A REP function returns the machine representation of a data object. It converts a data object to a bit string whose size is the actual number of bits occupied by the object.

The form is:

REP (name)

A REP function may be applied to named variables only. It may not be applied to tables with * bounds or to entries in parallel tables. A REP function may be used as a pseudo-variable.

REP Function -- Examples

1) Given the declaration:

ITEM SPEED U = 2;

If BITSINWORD is 16 and all items are allocated a full word,

REP(SPEED) yields 1B'0000000000000010'

Given the declaration:

ITEM NAME B 16 - 4B'FFFF';

If BITSINWORD is 9, BITSINBYTE is 4, and F is represented as 1B'1111'. Each word contains two bytes as follows:

xxxxyyyy0	where x indicates the first byte,
	y indicates the second byte, and
	0 indicates a filler bit. (Placement of filler bits is implementation dependent.)
REP(NAME	yields 1B'111111110111111110'
(*B 16*)(NAME)	yields 1B'1111111111111111'

2) BITSINWORD is 9, BITSINBYTE is 4, and F is represented as 1B'1111';

ITEM NAME B 16 = 4B'FFFF';

The assignment-statement:

```
REP(NAME) = 3B'750750';
```

changes the value of NAME from 1B'111111110111111110'
to 1B'111101000111101000';

SECTION 8

SHIFT FUNCTIONS

SOFTECH

THE SHIFT FUNCTIONS

The shift functions perform a logical shift of a bit formula. Two shift functions are defined, one for left shifting and one for right shifting.

The forms are:

SHIFTL (bit-formula , shift-count) for left shift

SHIFTR (bit-formula , shift-count) for right shift

The value returned by the shift functions is the bit-formula shifted as many positions as specified by shift-count. Its type is the same as the type of bit-formula.

<u>Example</u>

```
ITEM SUBMASK B 4 = 1B'1101';

    SHIFTL(SUBMASK, 2) returns 1B'0100'

    SHIFTR(SUBMASK, 3) returns 1B'0001'
```

NOTES: A logical shift loses the bits that are shifted out and fills vacated bits with zeros. Shift-count must be a non-negative integer formula. A shift-count of zero means no shift.

<u>Example</u>

```
SUB = SHIFTR (SUBMASK, 3);
```

SHIFT-FUNCTIONS -- EXERCISES

Given the declarations:

ITEM MASK B 10 = 1B'0101010101';

ITEM CODE B 6 = 3B'77';

Mark each of the following function calls correct(c) or incorrect (I). Assuming the items have their initialized values, give the result and type for those functions that are correct.

Function call	C	I	Result	Type
SHIFTL(MASK,5)				
SHIFTL(MASK,-2)				
SHIFTR(CODE,0)				
SHIFTR(MASK,3)				

ANSWERS

Given the declarations:

ITEM MASK B 10 = 1B'0101010101';

ITEM CODE B 6 = 3B'77';

Function call	C	I	Result	Type
SHIFTL(MASK,5)	X		1B'1010100000'	B 10
SHIFTL(MASK,-2)		X		
SHIFTR(CODE,0)	X		3B'77'	B 6
SHIFTR(MASK,3)	X		1B'0000101010'	B 10

SECTION 9

SIZE FUNCTIONS

SIZE FUNCTIONS

The size functions return the logical size of the argument. The functions are:

- BITSIZE (size-argument) returns number of bits
- BYTESIZE (size-argument) returns number of bytes
- WORDSIZE (size-argument) returns number of words

All partially filled bytes and words are included in the values returned by BYTESIZE and WORDSIZE respectively.

The value returned by BITSIZE is defined for each of the data types that can be used as a size-argument.

The result of the BITSIZE function depends on the type of the argument.

Argument Type	Result
U n	n (n = integer size)
S n	n + 1
F p	Number of bits actually occupied (this includes sign, exponent, and mantissa bits)
A scale, Fraction	scale + fraction + 1
B n	n
C n	n*BITSINBYTE
P	BITSINPOINTER
STATUS	Status-size if specified; otherwise, minimum number of bits necessary to represent status object of that type
TABLE	Number of bits from first bit of first entry to last bit of last entry
BLOCK	Number of words occupied by block * BITSINWORD

Examples

Descriptions*	Function Call	Result
ITEM RANGE S 10;	BITSIZE (RANGE)	11
ITEM POSITION U;	BITSIZE (POSITION)	15
ITEM AZIMUTH F 30;	BITSIZE (AZIMUTH)	actual number of bits
ITEM VELOCITY F;	BITSIZE (VELOCITY)	actual number of bits
ITEM SUBTOTAL A 6,2;	BITSIZE (SUBTOTAL)	9
ITEM MASK B 10;	BITSIZE (MASK)	10
ITEM FLAG B;	BITSIZE (FLAG)	1
ITEM ADDRESS C 26;	BITSIZE (ADDRESS)	26*8
ITEM CODE C;	BITSIZE (CODE)	1*8
ITEM LETTER STATUS (V(A), V(B), V(C), V(D), V(E), V(F), V(G), V(H);	BITSIZE (LETTER)	3
ITEM SWITCH STATUS V(ON), V(OFF));	BITSIZE (SWITCH)	1
	BYTESIZE (POSITION)	2
	WORDSIZE (POSITION)	1
	BITSIZE (MASK AND FLAG)	10

```
TABLE ATTENDANCE
(1:10) T 6;
   ITEM COUNT U 5
TABLE CONDITION(20) T;
  BEGIN
  ITEM ALERT B;
  ITEM CONTROL B;
  END
TABLE SPECIFICATIONS (99);
  BEGIN
  ITEM LENGTH U 5;
  ITEM WIDTH U 9;
  ITEM HEIGHT U 5;
  END
```

BITSIZE (ALERT(I))	1
(CONDITION(I))	2
(ATTENDANCE(I))	6
(SPECIFICATIONS)	3*100*16

NOTE: Table ATTENDANCE has tight structure with 6 bits-per-entry. Table CONDITION has tight structure, the default entry-size is 2.

*Assume BITSINWORD is 16, BITSINBYTE is 8.

Descriptions*	Function Call	Result
BLOCK GROUP; BEGIN ITEM COUNT U; ITEM VELOCITY F; TABLE TIMES(99); ITEM SECONDS U; END	BITSIZE (GROUP)	102*16

```
BLOCK GROUP;
  BEGIN
  ITEM COUNT U;
  ITEM VELOCITY F;
  TABLE TIMES(99);
    ITEM SECONDS U;
  END
```

NOTE: The block GROUP occupies 102 words.

*Assume BITSINWORD is 16, BITSINBYTE is 8.

SECTION 10

THE NWDSEN FUNCTION

THE NWDSEN FUNCTION

The NWDSEN function returns the number of words of storage allocated to each entry in the table or table-type-name given as its argument.

The form is:

NWDSEN (argument)

The type returned is S with default size.

Examples

```
TYPE DIMENSIONS TABLE;
        BEGIN
        ITEM LENGTH U;
        ITEM HEIGHT U;
        ITEM WIDTH U;
        END
TABLE COMPONENTS (10, 5) DIMENSIONS;
NWDSEN(DIMENSIONS) returns 3
NWDSEN(COMPONENTS) returns 3
TABLE COMPONENTS (10, 5) DIMENSIONS = 11 * 6 * NWDSEN
(DIMENSIONS) (0);
```

SECTION 11

STATUS INVERSE FUNCTION

SOFTECH

THE STATUS INVERSE FUNCTION

The status inverse functions find the lowest and highest values of the status-list associated with their argument.

The status inverse function that finds the lowest value has the form:

FIRST (argument)

The status inverse function that find the highest value has the form:

LAST (argument)

The type of the result is the same as the type of the argument.

<u>Examples</u>

Given the following declarations:

```
ITEM LETTER STATUS
    (V(A),V(B),V(C),V(D),V(E),V(F),V(G),V(H));
ITEM SWITCH STATUS
    (V(ON),V(OFF));
```

The following functions have the following results:

Function Call	Function Value
FIRST(LETTER)	V(A)
LAST(LETTER)	V(H)
FIRST(SWITCH)	V(ON)

STATUS INVERSE FUNCTIONS -- EXERCISES

Given the following declarations:

```
TYPE SEASON STATUS (14V(SPRING), 7V(SUMMER), V(FALL),
    3V(WINTER));
TYPE COLOR STATUS (V(RED),V(ORANGE),V(YELLOW),
    V(GREEN),V(BLUE),V(VIOLET);
```

Give the values of the following function calls:

Function Call	Result
FIRST(SEASON)	
LAST(SEASON)	
FIRST(COLOR)	

ANSWERS

Given the following declarations:

```
TYPE SEASON STATUS (14V(SPRING), 7V(SUMMER), V(FALL),
      3V(WINTER));
TYPE COLOR STATUS(V(RED),V(ORANGE),V(YELLOW),
      V(GREEN),V(BLUE),V(VIOLET);
```

Give the values of the following function calls:

Function Call	Result
FIRST(SEASON)	V(WINTER)
LAST(SEASON)	V(SPRING)
FIRST(COLOR)	V(RED)

SECTION 12

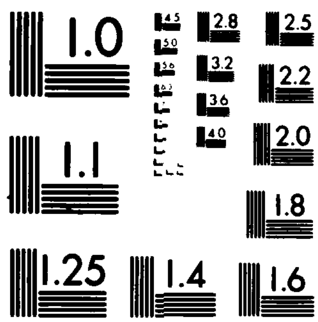
THE NEXT FUNCTION

SOFTECH

AD-A108 527 SOFTECH INC WALTHAM MA F/G 5/9
THE JOVIAL (J73) WORKBOOK. VOLUME I. INTEGER AND FLOATING POINT--ETC(U)
NOV 81 F30602-79-C-0040
UNCLASSIFIED RADC-TR-81-333-VOL-1 NL

5 of 5

END
DATE FILMED
1-82
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

THE NEXT FUNCTION

The NEXT function may obtain a successor or predecessor of a status argument. The form is:

NEXT (argument, increment)

The NEXT function returns a status value.

NOTES: Increment is an integer formula. If increment is positive, NEXT returns a successor. If increment is negative, NEXT returns a predecessor. Increment may not select a value outside the range of the status-list. The status-list must have default representation. Argument may not be an ambiguous status-constant.

Example

```
TYPE SHAPE STATUS (V(A), V(B), V(C), (V(D), V(E), V(F);
ITEM JJ SHAPE;
.
.
.
FOR JJ : FIRST (SHAPE) THEN NEXT (JJ, 1) WHILE JJ < LAST
        (SHAPE);
        .
        .
        .
```

THE NEXT FUNCTION -- POINTER

The NEXT function may obtain the arithmetic sum of a pointer argument and an increment. The form is:

NEXT (argument, increment)

The NEXT function returns a pointer.

> *NOTES: Increment is an integer formula. The value returned is:*
>
> *pointer-formula + increment * LOCSINWORD*
>
> *The programmer must take care that when using NEXT with a pointer argument, the pointer points to a programmer defined object.*

THE NEXT FUNCTION -- EXERCISES

Given the following declarations:

```
TYPE SEASON STATUS (V(SPRING),V(SUMMER),V(FALL),
       V(WINTER);
ITEM CLIMATE SEASON = V(SUMMER);
TYPE CITATION
       TABLE (1000);
              BEGIN
              ITEM NAME C 20;
              ITEM TITLE C 50;
              ITEM DATE C 6;
              END
TABLE MATHEMATICS CITATION;
ITEM CITATION'PTR P CITATION = LOC(MATHEMATICS);
```

Assuming that the data objectives have their initialized values and that LOC(MATHEMATICS) is 25600, give the results of the following function calls:

Function Call	Result
NEXT(CLIMATE,1)	
NEXT(CLIMATE,-1)	
NEXT(CLIMATE,0)	
NEXT(CITATION'PTR,2)	

ANSWERS

Given the following declarations:

```
TYPE SEASON STATUS (V(SPRING),V(SUMMER),V(FALL),
      V(WINTER);
ITEM CLIMATE SEASON = V(SUMMER);
TYPE CITATION
      TABLE (1000);
            BEGIN
            ITEM NAME C 20;
            ITEM TITLE C 50;
            ITEM DATE C 6;
            END
TABLE MATHEMATICS CITATION;
ITEM CITATION'PTR P CITATION = LOC(MATHEMATICS);
```

Assuming that the data objectives have their initialized values and that LOC(MATHEMATICS) is 25600, give the results of the following function calls:

Function Call	Result
NEXT(CLIMATE,1)	V(FALL)
NEXT(CLIMATE,-1)	V(SPRING)
NEXT(CLIMATE,0)	V(SUMMER)
NEXT(CITATION'PTR,2)	25600 + 2 * LOCSINWORD

SECTION 13

THE BOUNDS FUNCTION

SOFTECH

THE BOUNDS FUNCTION

The bounds functions find the lower or upper bound of a table dimension.

The function to find the lower-bound has the form:

LBOUND(argument, dimension-number)

The function to find the upper-bound has the form:

UBOUND(argument , dimension-number)

Argument may be a table-name or a table-type-name. The type returned is either integer or status depending on the type of the dimension.

> *NOTES: Dimension-number is an integer formula known at compile-time. Dimensions are numbered left to right starting with zero. For parameters declared with asterisk bounds, LBOUND always returns 0.*

Examples

1) TABLE TABNAME(1:5, 2:17, 8, 1:7);

LBOUND (TABNAME, 1)	returns 2
UBOUND (TABNAME, 3)	returns 7
LBOUND (TABNAME, 2)	returns 0

2)
```
FOR I:LBOUND(TABNAME,0) BY 1 WHILE I <= UBOUND(TABNAME,0);
   FOR J:LBOUND(TABNAME,1) BY 1 WHILE J <= UBOUND
   (TABNAME,1);
      FOR K:0 BY 1 WHILE K <= UBOUND(TABNAME,2);
         FOR L;LBOUND(TABNAME,3) BY 1 WHILE L <=
         UBOUND(TABNAME,3);
            TABENT(I,J,K,L) = 0;
```

3) PROC CLEAR (: TABNAME);

```
BEGIN
TABLE TABNAME (*, *);
    ITEM NUMBER S;
FOR I : 0 BY 1 WHILE I <= UBOUND (TABNAME, 0);
    FOR J : 0 BY 1 WHILE J <= UBOUND (TABNAME, 1);
        NUMBER (I, J) = 0;
END
```

NOTES: Asterisk LBOUNDS always are zeros.

SECTION 14

SUMMARY OF BUILT-IN FUNCTIONS

SOFTECH

SUMMARY OF BUILT-IN FUNCTIONS

LOC (argument)

The LOC function obtains the machine address of its argument.

ABS (numeric-formula)

The ABS function returns the absolute value of the numberic-formula.

SGN (numeric-formula)

The sign function returns an indication of the sign of the numberic argument.

BIT (bit-formula, first-bit, length)

The BIT function selects a substring from a bit-formula.

BYTE (character-formula, first-character, length)

The BYTE function selects a substring from a character formula.

REP (name)

The REP function obtains the machine representation of a data object.

SHIFTR (bit-formula, shift-count)
SHIFTL (bit-formula, shift-count)

The shift right and shift left functions perform a logical shift on a bit string.

BITSIZE (size-argument)
BYTESIZE (size-argument)
WORDSIZE (size-argument)

The BITSIZE, BYTESIZE and WORDSIZE functions return the logical size of a data argument in full bits, bytes, and words.

NWDSEN (argument)

The NWDSEN function returns the number of words in an entry in a table.

FIRST (argument)

LAST (argument)

The FIRST and LAST functions return the status-constant with the smallest and largest representation in a given status-argument.

NEXT (argument)

The NEXT function returns the successor or predecessor of its status argument, or the incremented machine address of its pointer argument.

UBOUND (argument, dimension-number)

LBOUND (argument, dimension-number)

The U-BOUND and L-BOUND functions return the upper and lower bounds of a given table for given dimensions.

SUMMARY OF BUILT-IN FUNCTIONS -- EXERCISE

```
TYPE COLOR STATUS (V(BROWN), V(BLUE), V(GREEN), V(GREY));
TYPE PERSONNEL
        TABLE (1:40);
                BEGIN
                ITEM HEIGHT A 10,4;
                ITEM WEIGHT F 20;
                ITEM EYES COLOR;
                ITEM HAIR COLOR;
                ITEM FIRSTNAME C 15;
                ITEM LASTNAME C 25;
                ITEM MARRIED B;
                END
ITEM PERSPTR P PERSONNEL;
TABLE PERSON PERSONNEL;
ITEM PINX U = 3;
ITEM $BIT1 B 16 = 2B'1';
ITEM $BIT2 B 16 = 4B'1';
ITEM $CHAR C 12 = 'GANG BUSTER!';
```

Assume the following implementation parameters:

BITSINBYTE	8
BYTSINWORD	32
BYTESINWORD	4

Using the declarations and assumptions given on the previous page, give values returned by the following built-in functions.

Function	Value
1. BYTESIZE($CHAR)	
2. LOC(PERSON)	
3. BYTE($CHAR,6,2)	
4. (assuming HAIR-V(BLUE)) NEXT(HAIR,2)	
5. SHIFTR($BIT2,12)	
6. UBOUND(PERSON,0)	
7. LBOUND(PERSON,0)	
8. BITSIZE(HEIGHT(PINX))	
9. BIT($BIT1,15,1)	
10. BYTE($CHAR,0,4)	
11. BITSIZE(EYES(8))	
12. NEXT(LAST(EYES(8)),-2)	
13. SGN(LBOUND(PERSON,0))	
14. SHIFTL($BIT2,2)	
15. NWDSEN(PERSON)	
16. WORDSIZE($CHAR)	
17. BITSIZE(PERSON)	

ANSWERS

Function	Value
1. BYTESIZE($CHAR)	12
2. LOC(PERSON)	Pointer of type PERSONNEL containing address of PERSON.
3. BYTE($CHAR,6,2)	'US '
4. (assuming HAIR-V(BLUE)) NEXT(HAIR,2)	V(GREY)
5. SHIFTR($BIT2,12)	1B'0000000000000000'
6. UBOUND(PERSON,0)	40
7. LBOUND(PERSON,0)	1
8. BITSIZE(HEIGHT(PINX))	15
9. BIT($BIT1,15,1)	1B'0000000000000001'
10. BYTE($CHAR,0,4)	'GANG '
11. BITSIZE(EYES(8))	2
12. NEXT(LAST(EYES(8)),-2)	V(BLUE)
13. SGN(LBOUND(PERSON,0))	+1
14. SHIFTL($BIT2,2)	1B'0000000000000100'
15. NWDSEN(PERSON)	16
16. WORDSIZE($CHAR)	3
17. BITSIZE(PERSON)	16*32*40

BLOCK-PRESETS

Blocks may be preset. If a block-preset is given on a block declared using a block type-name, it must be given as part of the block-heading. Otherwise, it is given as part of the block-body.

Examples

a.

```
BLOCK GETDATA;
    BEGIN
    ITEM NAME C 20 = 'J. JONES';
    TABLE REPORT (1 : 4) = V(GOOD), 'B';
        BEGIN
        ITEM ATTITUDE STATUS (V(POOR), V(AVE),
          V(GOOD));
        ITEM GRADE C 1;
        END
    END
```

b. Using the TYPE BLOCK GROUP, as declared on the previous page, the following block-declaration with a block preset can be made:

```
BLOCK MAPPINGS GROUP = 3, (20 (3.2, -3.2)), ('XXX,
   (12 (6)));
```

DECLARATIONS -- EXERCISES

Write JOVIAL (J73) declarations for the following:

1. A counter to be used to count each day of the course as it passes.

2. A variable to reference a person's monthly income tax deductions. This variable must always be accurate to two decimal places.

3. A type that may contain your full name.

4. A type that may take an integer values from -20 through 20.

5. A variable to be used to indicate an emotional state.

6. A variable to keep track of every time a subroutine is called.

7. A group of items to describe the information one must give when captured. (Use the type declared for #3 where appropriate.)

8. A group of items to describe a checker board. The black squares (treat as every other square) should be preset to -16. (Use the type declared for #4 where appropriate.)

9. A collection of the names of students in this class and the number of students. (Use the type declared for #4 where appropriate.)

10. A collection of information on your paycheck stub. This may include a group of deductions, a group of kinds of pay, name, amount, etc.

ANSWERS

```
1.    ITEM COUNT U;
2.    ITEM DEDUCT A 20, 7;
3.    TYPE FULL'NAME C 50;
4.    TYPE VALUE S 5;
5.    ITEM EMOT STATUS (V(HAPPY), V(SAD));
6.    ITEM COUNTER U;
7.    TABLE CAPTURED;
          BEGIN
          ITEM NAME FULL'NAME;
          ITEM RANK C 9;
          ITEM SERIAL'NO C 9;
          END
8.    TABLE CHECKERS (1 :32);             TABLE CHEX (1 :8, 1: 8);
          BEGIN                               ITEM SQUARE S =
          ITEM BLACK VALUE = 32(-16);           4(4(-16),
          ITEM WHITE VALUE;                       (4(, -16));
          END
9.    BLOCK CLASS;
          BEGIN
          ITEM NUMBER VALUE;
          TABLE NAMES (1 : 30);
              ITEM PERSON C 20;
          END
```

10. BLOCK PAYCHECK;

```
BEGIN
ITEM NAME C 20;
ITEM SSN C 9;
TABLE DEDUCTIONS;
    BEGIN
    ITEM FICA A 20, 7;
    ITEM FED'INC A 20, 7;
    ITEM STATE'INC A 20, 7;
    END
END
```

SECTION 4

SUMMARY

TABLE-DECLARATIONS

A table declares a group of objects. The form of a table-declaration is:

[CONSTANT] TABLE name [table-attributes] ;

entry-description

or

TABLE name [table-attributes] item-type-description
[table-preset] ;

Entry-description may either by simple:

item-declaration

or compound:

BEGIN

item-declaration...

END

Table-attributes shown so far are:

[dimension-list] [table-type-name] [table-preset]

Dimension-list is of the form:

(dimension, ...)

Dimension is of the form:

[lower-bound :] upper-bound

NOTES: A dimension-list may have as many as seven dimensions. Lower-bound and upper bound must both be of type S, U, or STATUS. Lower-bound must be less than or equal to upper-bound. If lower-bound is not specified, the default lower-bound is:

- *zero if upper-bound is of type S or U.*
- *the first status-constant if upper-bound is of type STATUS.*

TABLE TYPE-DECLARATIONS

A table-type-name may be declared to describe a table type.

A simple form of a table type-declaration is:

TYPE table-type-name TABLE ;

entry-description

A table-type-name may then be used in a table-declaration:

TABLE name table-type-name [table-preset] ;

> *NOTES: A type-declaration does not declare any data; table-presets may not appear. Presets may only appear on the heading of the table-declaration.*
>
> *A table-declaration using a table-type-name must not have an entry-description.*

LIKE OPTIONS

A table type-declaration may include a like-option:

TYPE table-type-name TABLE like-option;

entry-description

A like-option is of the form:

LIKE table-type-name

> *NOTES: A like-option allows the program to create a variety of tables with a common front part.*
>
> *A like-option is used only in a table type-declaration. A like-option must not be used in a table-declaration.*

DIMENSIONS

A table type-declaration may include a dimension list:

TYPE table-type-name TABLE dimension-list ;

A more complete form of a table type-declaration is:

TYPE table-type-name TABLE [dimension-list] [like-option] ;

entry-description

NOTES: A table-type declaration may have one and only one dimension-list. If the like-option refers to a dimensioned table type, the type-declaration itself may not have a dimension-list. If the type-description has a dimension-list, a table being declared with that table-type-name must not also have a dimension-list.

CONSTANT TABLE-DECLARATIONS

A table may be declared to be a constant table. The form is:

CONSTANT TABLE name [table-attributes] ;

NOTES: Not all items in all entries of a constant table need to be preset. *The table-preset may be given as a part of the table-attributes or as a part of the entry-description.*

TABLE PRESETS

A table-preset initializes items within an entry of a table. A table-preset may be given as a part of the entry-description or as a part of the table-attributes. The form is:

= value, ...

Values may be omitted in a table-preset to indicate the corresponding item in an entry will not be preset. Values may be omitted within the list, (by placing commas next to each other), or at the end of the list, (by use of a semi-colon to end the declaration).

A value or a group of values may be repeated. The form is:

repetition (preset-option, ...)

A preset-option has the form:

$$\left\{\begin{array}{l}\text{value}\\ \text{repetition (preset-option, ...)}\end{array}\right\}$$

A positioner may be used to select a specific entry to preset. The form is:

POS (index-list) : value, ...

Index-list has the form:

index, ...

> *NOTES: The indices used in a positioner must agree in number and in type with the dimensions. The values of the indices must lie within the ranges prescribed by the dimensions.*

TYPE EQUIVALENCE, IMPLICIT CONVERSION

Two tables are equivalent if they have the same number of dimensions, same number of elements in each dimension, same number of items in each entry, and the type and order of the items are the same.

No implicit conversions are performed.

Example

```
TABLE ONE (1 : 10, 2 : 5, 2 : 3);      TYPE SWITCH STATUS (V(ON), V(OFF));
         BEGIN                         TABLE TWO (81 : 90, 3, V(ON) : V(OFF);
         ITEM AA U;                                BEGIN
         ITEM BB F;                                ITEM CC U;
         END                                       ITEM DD F;
                                                   END
```

Tables ONE and TWO are EQUIVALENT: they both have three dimensions; they both have ten entries in the first dimension, four entries in the second dimension, and two entries in the third dimension; each entry has two items; and the first item is of type U and the second is of type F.

EXPLICIT CONVERSION

A bit string may be converted to the value of a table type if the size of the bit string is equal to the BITSIZE of the table type.

A table-conversion may be applied to a table object of that type merely to clarify (for the reader) the type of that object. It may not change the type of the table.

Example

```
TYPE XX TABLE;
     ITEM XX1 U;
TABLE YY XX;
     ... (* XX *) (YY) ...
```

BLOCK DECLARATIONS

A block is a collection of items, tables, and nested blocks. A block declaration has the form:

BLOCK *name;*

block-body

A block body is:

$$\left\{ \begin{array}{l} \text{item-declaration} \\ \text{table-declaration} \\ \text{block-declaration} \end{array} \right\} \ldots$$

A block type-declaration declares a block-type-name that may be used in a block-declaration. The form is:

TYPE block-type-name BLOCK

block-body

The block type-name may then be used in a block-declaration:

BLOCK name block-type-name [block-preset];

Blocks may be preset. If a block-preset is given on a block declared using a block type-name, it must be given as part of the block-heading. Otherwise, it is given as part of the block-body.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.


FILMED
8